RL-TR-92-52, Vol II (of two)
Final Technical Report
April 1992

SOFTWARE RELIABILITY, MEASUREMENT, AND TESTING Guidebook for Software Reliability Measurement and Testing

Science Applications International Corp. (SAIC)
Research Triangle Institute (RTI)

James A. McCall and William Randell (SAIC)
Janet Dunham and Linda Lauterbach (RTI)

Reproduced From
Best Available Copy

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

Rome Laboratory
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-92-52, Vol II (of two) has been reviewed and is approved for publication.

APPROVED:

Joseph P Cavano

JOSEPH P. CAVANO
Project Engineer

FOR THE COMMANDER:

Raymond P. Urtz Jr

RAYMOND P. URTZ, JR.
Technical Director
Command, Control, & Communications Directorate

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL(C3CB), Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)

2. REPORT DATE
April 1992

3. REPORT TYPE AND DATES COVERED
Final Sep 86 - Dec 89

4. TITLE AND SUBTITLE
SOFTWARE RELIABILITY, MEASUREMENT, AND TESTING
Guidebook for Software Reliability Measurement and Testing

5. FUNDING NUMBERS
C - F30602-86-C-0269
PE - 62702F
PR - 5581
TA - 20
WU - 63

6. AUTHOR(S)
James A. McCall and William Randell (SAIC)
Janet Dunham and Linda Lauterbach (RTI)

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Science Applications International Corp. (SAIC)
10260 Campus Point Drive, San Diego CA 92121

Research Triangle Institute (RTI)
PO Box 12194, Research Triangle Park NC 27709

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Rome Laboratory (C3CB)
Griffiss AFB NY 13441-5700

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
RL-TR-92-52, Vol II (of two)

11. SUPPLEMENTARY NOTES
Rome Laboratory Project Engineer: Joseph P. Cavano/C3CB/(315) 330-4063

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This effort integrated software reliability, measurement, and test techniques in terms of prediction, estimation, and assessment. Experiments were conducted to compare six testing techniques and to measure the effect of software product and process variables on software reliability. A guidebook was produced to help program managers control and manage software reliability and testing. Error/anomaly and code reviews were the test techniques found to be the most effective at the unit level; branch testing and code reviews were the most effective at the CSC level.

NOTE: Rome Laboratory/RL (formerly Rome Air Development Center/RADC)

14. SUBJECT TERMS
Software Reliability, Software Measurement, Software Testing

15. NUMBER OF PAGES
256

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT
UNCLASSIFIED

18. SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

19. SECURITY CLASSIFICATION OF ABSTRACT
UNCLASSIFIED

20. LIMITATION OF ABSTRACT
UL

TABLE OF CONTENTS

TABLE OF CONTENTS (Cont.)

LIST OF FIGURES

LIST OF APPENDICES

EVALUATION

The concepts needed to understand reliability are not fully developed. Many of the important issues are too broad for a single-focused treatment and must be explored from different angles. This is especially true for undertaking software reliability because software is intangible and difficult to understand in its own right - adding reliability only complicates the subject.

The goal of this report is to help bridge the gap between what management can control and what production needs to do. Although the book addresses software reliability and testing, the approach could be generalized to producing quality in other domains. More precisely, this report lays the foundation for the bridge to software reliability - it attempts to show how to quantitatively evaluate the process of developing high reliability software so that one can improve upon the process in the future. The prediction and estimation numbers produced for software reliability are more valuable for comparing with other projects and in tracking progress toward continual quality improvement than in their absolute values. Further research in experimentally applying reliability measures across software development, review and test processes is necessary to validate the numbers. If you are not interested in improving quality on a long-term basis, then this report may not be especially helpful.

Although the measures described in this report produce exact software reliability numbers for fault density or failure rate, these numbers must be used with discretion because the proper relationship with specific levels of reliability have not yet been proven. there is no magical formula for deriving reliability predictions or assessments because software engineering does not yet have the necessary theory upon which to develop such equations. This does not mean that empirical observations cannot be used to develop a discipline. Instead, the reader is cautioned that the empirical observations made during this effort were not extensive enough to prove their validity over all projects. For example, the measures related to the development environment try to relate various characteristics of an environment to their individual impact on reliability. In the projects studied, there was not enough diversity in these characteristics to enable exact predictions at that level. Although data analyses lead to equations, the results are not appropriate across the complete range of possible outcomes - in fact, low values for the 'Dc' metric produce erroneous negative numbers. It is better to treat this metric at a more global level (i.e., organic, semi-detached or embedded) as shown in Metric Worksheet 1A.

In facing such situations, choices had to be made between the theoretical or ideal state and providing suggestions on how a typical organization could customize, develop and use reliability measures, tailored to their unique procedures. This report leans to the practical side of measurement by showing the role that reliability prediction and estimation could play in the future. Purists might be disappointed in this.

The path to higher reliability and better testing is not always easy. To improve a process, change is required. If you don't intend to change your current procedures, this report may not be of much value On the other hand, if you plan to experimentally apply the techniques described in this report, consistently observe results over several projects, and tailor the techniques and measures for your organization, then, hopefully, you will see an improvement in your software's reliability.

Joseph P. Cavano

PREFACE

This Guidebook contains the results of a research and development effort by Science Applications International Corporation (SAIC) and Research Triangle Institute (RTI) to integrate and improve the application of software reliability measurement and testing techniques. This Guidebook is part of the Final Report of the project. This effort was performed under Contract Number F30602-86-C-0269 for the U.S. Air Force Rome Air Development Center (RADC).

Refinements to the present RADC methodologies for software reliability and testing, and recommendations resulting from the experiments and empirical studies were incorporated into this RADC Software Reliability Measurement and Testing Guidebook for Air Force acquisition managers. This new Guidebook represents the integration and updating of two existing Guidebooks: RADC Software Test Handbook (STH), RADC TR 84-53 and the RADC Software Reliability Prediction and Estimation Guidebook (SRPEG), RADC TR 87-171. The revised and integrated Guidebook provides guidance on selecting state-of-the-practice testing techniques, and provides instructions for collecting metric data on software development projects and analyzing the data to predict and estimate the future reliability of the final product.

1.0 INTRODUCTION

1.1 Purpose

This Guidebook provides procedures for both the preparation of software reliability predictions and estimations and for planning the testing of embedded and separately procured Air Force computer systems.

The results of prediction and estimation are primarily intended to serve as relative indicators of reliability in connection with design decisions and in monitoring progress of a project. Caution must be used in equating predicted or estimated values of software reliability with operational values, as is also the case in hardware reliability prediction.

The procedures for software testing are designed to assist Air Force software developers and maintainers in the selection and effective use of higher order language (HOL) software testing techniques and in the application of automated tools for the testing of computer programs. Software reliability estimations are derived from test results and so are closely integrated with the software testing techniques.

The guidebook also includes typical paragraphs than can be included in Air Force software development statements-of-work to specify the use of advanced software reliability and testing techniques by the contractor during the analysis, design, coding and testing and verification phases of a contracted development.

1.2 Application

The requirements and procedures established by this Guidebook may be selectively applied to any Department of Defense contract-definitized procurements, request for proposals, statements of work, and in-house Government projects for system development and production. It is not intended that all the requirements herein will need to be applied to every program or program phase. Procuring activities shall tailor the requirements of this standard to the minimum needs of each procurement and shall encourage contractors to submit cost effective tailoring recommendations.

Guidelines provided in the Guidebook can be applied during the computer software analysis, design, coding, and test and integration phases of development test and evaluation (DT&E), as defined in DoD-STD-2167A.

The method by which the user prepares software reliability predictions and estimations are based on and supersede the Software Reliability Prediction and Estimation Guidebook (SRPEG).

The method by which the user selects testing techniques is based on and updates the RADC Software Test Handbook (STH). The methodology has basically remained the same assisting the user to select the most appropriate testing techniques based on the use of rating tables. A number of these tables have been revised based on the data collected during an RADC funded experiment. They permit a compact representation of many considerations and recommendations which result from formal experiments and studies in software testing and in the use of modern testing techniques and automated test tools. The testing techniques recommendations are based on quantitative and qualitative information and should be regarded as guidelines, not as rigid rules.

1.3 Scope

Software reliability prediction and estimation techniques are described as a methodology in this guidebook for assessing a software system's ability to meet specified reliability requirements. Software reliability prediction translates software measurements taken during early life cycle phases, into a predicted reliability. Software reliability estimation, based on test phase indicators, estimates how reliably the software will perform its required functions in its operational environment. When used in combination, these software reliability and testing techniques provide a basis for identifying areas wherein special emphasis or attention is needed, for comparing the cost-effectiveness of various design configurations, and for evaluating correct execution of the developed software. This guidebook is intended as a companion document to MIL-STD-785B, MIL-STD-756B, MIL-HDBK 217E and DoD-STD-2167A.

The Test Guidelines identified in this guidebook and further described in the Software Test Handbook, RADC TR 84-53, can be applied during the computer software coding and checkout, test and integration, and operation and support phases of development test and evaluation (DT&E), operational test and evaluation (OT&E) and verification and validation (V&V), as defined in AFR 80-14, "Research and Development Test and Evaluation", and AFR 800-14, Vol. II, "Acquisition and Support Procedures for Computer Resources in Systems".

1.4 Organization of Guidebook

This guidebook is organized as follows:

Section 1 provides an introduction and description of the application and scope of the guidebook.

Section 2 provides a list of applicable documents.

Section 3 describes the general requirements of software reliability, identifies the components of a Software Reliability Program, and introduces the procedures described in the Task Sections.

TASK SECTION 100	Reliability Prediction Task
TASK SECTION 200	Software Test Technique Selection Task
TASK SECTION 300	Reliability Estimation Task
APPENDIX A	Definition and Terminology
APPENDIX B	Procedures and Worksheets
APPENDIX C	Answer Sheets
APPENDIX D	Software Testing Technique Tools

2.0 REFERENCED DOCUMENTS

2.1 Issues of Documents

The following documents are referenced in this guideline for information and guidance. The issue in effect on date of invitation for bids or request for proposal should be used by the acquisition agency.

STANDARDS

MIL-STD-785B	Reliability Program for Systems and Equipment Development and Production
MIL-STD-721	Definitions of Terms for Reliability and Maintainability
MIL-STD-781C	Reliability Design Qualification and Production Acceptance Tests: Exponential Distribution
MIL-STD-105	Sampling Procedures and Tables for Inspection by Attribute
MIL-STD-1521A	Technical Reviews and Audits for Systems, Equipment, and Computer Programs
MIL-HDBK-217E	Reliability Prediction of Electronic Equipment
MIL-STD-756B	Reliability Modeling and Prediction
MIL-STD-2167A	Defense System Software Development
MIL-STD-2168	Software Quality Evaluation (Proposed)
MIL-STD-1679	Weapon System Software Development
MIL-STD-490	Specification Practices
MIL-STD-480	Configuration, Control, Engineering Changes, Deviations, and Waivers
MIL-STD-483	Configuration Management Practices for System, Equipment, Munitions, and Computer Programs
MIL-Q-9858	Quality Program Requirements

2.2 Other Publications

The following documents are potential sources of reliability data or describe techniques that may be used in conjunction with this Guidebook. Specific requirements for use of these or other data sources must be specified by the procuring activity.

RADC TR 85-37	"Specification of Software Quality Attributes", Feb 85
RADC TR 85-228, Vol I	"Impact of Hardware/Software Faults on System Reliability - Study Results", Dec 85
RADC TR 85-228, Vol II	"Impact of Hardware/Software Faults on System Reliability - Procedures for use of Methodology, Dec 85
RADC TR 83-176	"A Guidebook for Software Reliability Assessment", 1983
RADC TR 84-53	"Software Test Handbook", Mar 84
RADC TR 87-171	"Software Reliability Prediction and Estimation Guidebook", May 87

3.0 GENERAL REQUIREMENTS

3.1 Software Reliability

Software reliability prediction, estimation, and testing shall be planned and performed in accordance with the general requirements of this guidebook and the task(s) and method(s) specified by the procuring activity.

3.1.1 The Reliability Problem

When it is proposed to design a system which includes computers to perform a complex and demanding job, it is assumed that the required investment will be justified according to the perfection by which the job is performed or by the large number of times which the system can do the job. This assumption cannot be justified when a system fails to perform upon demand or fails to perform repeatedly. Thus, the reliability of a system is critical to its cost effectiveness.

Reliability is a consideration at all levels of systems, from electronic components to operating systems to application software because the components are combined in systems of ever increasing complexity and sophistication. Therefore, at any level of development and design, it is natural to find the influence of reliability engineering acting as a discipline devoting special engineering attention to the unreliability problem. Reliability engineering has been primarily concerned with the time degradation of materials, physical and electronic measurements, equipment design, processes and system analysis, and synthesis. This Guidebook extends that discipline to software reliability engineering. None of these can be isolated from the overall electronics context or software development process but must be carried on in conjunction with many other disciplines.

Software Testing (Test planning) has been included in this guidebook because of its key role in identifying software reliability problems and providing data (failures and failure rates) for reliability estimation.

3.1.2 The Role of Reliability Prediction and Estimation and Testing in Software Engineering

To be of value, a prediction or estimation must be timely. However, the earlier it is needed, the more difficulties will be encountered. It is certainly true that the earlier a prediction has to be made about the unknown nature of a future event, the more difficult it is to make a meaningful prediction. As an example, it can be seen that the reliability of an electronic equipment is known with certainty after it has been used in the field and it is worn out and its failure history has been faithfully recorded. But, for purposes of doing anything about the reliability of this equipment, this knowledge has little value. Before this point, reliability cannot be known with certainty; but a great deal of knowledge about reliability can be accumulated over a short period early in the useful life. Even though the degree of certainty of knowledge is less, there is some opportunity to do something to influence the reliability of the remaining life portion.

Considering the various stages back through installation, shipment, test, production, test design, development, procurement, etc., less and less can be known with certainty about reliability the earlier in the life cycle you are. However, what is known or predicted becomes more and more valuable as a basis for taking action. After all, there is no value in simply knowing that a certain failure will occur at some specific time in the future. The value comes in having the opportunity to do something to prevent the failure from occurring. Once this is done, the future is changed from what was predicted with certainty. Thus, prediction becomes part of a process of "designing the future".

Figure 3-1 illustrates this concept in the context ot software development. This figure depicts the framework for software reliability that this guidebook is based on. It shows the four major software reliability activities covered by this guidebook: reliability specification, prediction, estimation, and assessment; and how those activities relate to DoD 2167A life cycle phases. Also shown are the basic metrics (i.e., measurement points) that provide the basis for predictions and estimations of reliability. Note reliability estimation utilizes data captured from the testing phases of the software development.

The roles predictions play are:

- early reliability predictions compared to the reliability goal of the system (specification) supports the analyses of alternative designs and architectural decisions.
- predictions at major milestones/reviews (e.g., PDR, CDR) support evaluation of the proposed design and facilitate identification of reliability shortfalls and rework requirements.
- predictions during code development indicate the need for standards improvement, rework requirements, and assistance in test planning.

The roles estimation plays are:

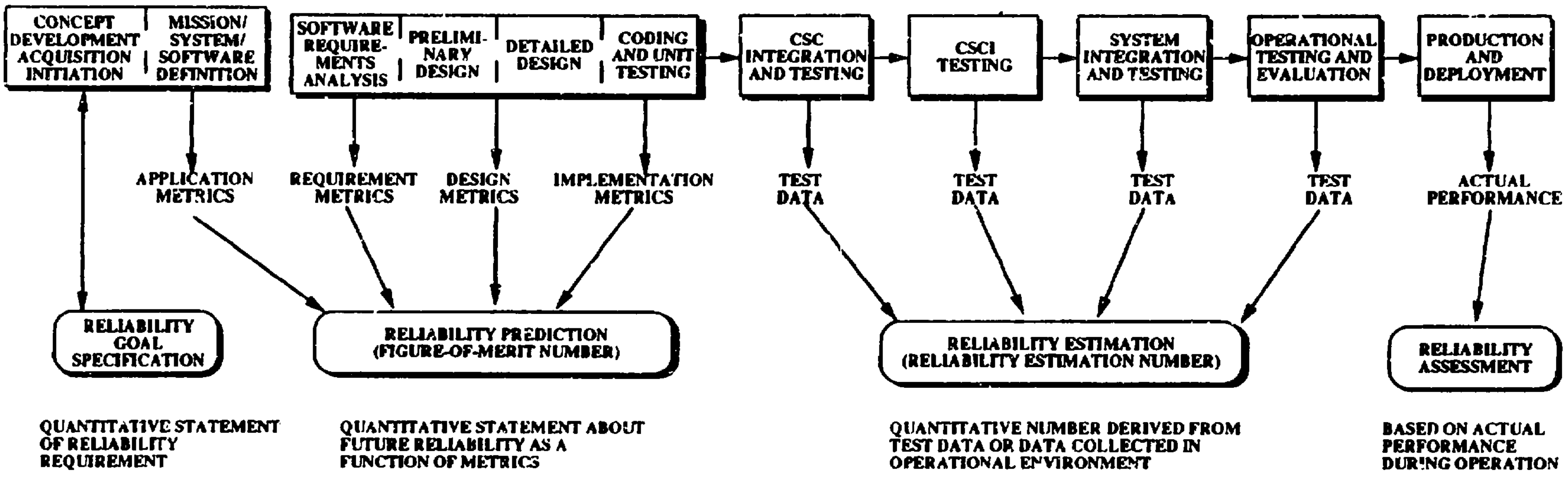
- estimations support comparison of software performance during test with reliability goals.
- estimations support on-going test planning, rework, and retest requirements.
- estimations support reliability growth monitoring.
- estimations support the decision process to proceed to the next phase of testing and acceptance testing.

The roles testing plays from a reliability viewpoint are:

- incremental phases of evaluation by test case conduct of individual units, CSC's, CSCI's and integration as a system.
- demonstration of completeness of requirements fulfillment.
- demonstration of acceptability (qualification) for production or delivery.

In general the software reliability concepts covered in this guidebook assist in:

- improving the overall quality of the product by early indications of problems and by more quantitative assessment of these problems.
- providing insight into quality versus cost and schedule tradeoffs, life cycle costs, software product warranty considerations, risk and liability, and performance expectations.
- improving test planning to aid in more thorough evaluation of the software prior to acceptance.


CONCEPT DEVELOPMENT ACQUISITION INITIATION
MISSION/ SYSTEM/ SOFTWARE DEFINITION
SOFTWARE REQUIRE-MENTS ANALYSIS
PRELIMI-NARY DESIGN
DETAILED DESIGN
CODING AND UNIT TESTING
CSC INTEGRATION AND TESTING
CSCI TESTING
SYSTEM INTEGRATION AND TESTING
OPERATIONAL TESTING AND EVALUATION
PRODUCTION AND DEPLOYMENT
APPLICATION METRICS
REQUIREMENT METRICS
DESIGN METRICS
IMPLEMENTATION METRICS
TEST DATA
TEST DATA
TEST DATA
TEST DATA
ACTUAL PERFORMANCE
RELIABILITY GOAL SPECIFICATION
RELIABILITY PREDICTION (FIGURE-OF-MERIT NUMBER)
RELIABILITY ESTIMATION (RELIABILITY ESTIMATION NUMBER)
RELIABILITY ASSESSMENT
QUANTITATIVE STATEMENT OF RELIABILITY REQUIREMENT
QUANTITATIVE STATEMENT ABOUT FUTURE RELIABILITY AS A FUNCTION OF METRICS
QUANTITATIVE NUMBER DERIVED FROM TEST DATA OR DATA COLLECTED IN OPERATIONAL ENVIRONMENT
BASED ON ACTUAL PERFORMANCE DURING OPERATION

Note: THE RELIABILITY FIGURE-OF-MERIT NUMBER AND THE RELIABILITY ESTIMATION NUMBERS MUST BE DEFINED IN THE SAME UNITS OF MEASUREMENT SO THAT THEY MAY BE MEANINGFULLY COMPARED AND RELATED TC EACH OTHER.

FIGURE 3-1 FRAMEWORK FOR SOFTWARE RELIABILITY

3.2 Software Reliability Program

The contractor shall establish and maintain an efficient reliability program to support economical achievement of overall program objectives. To be considered efficient, a reliability program shall clearly: (1) improve operational readiness and mission success of the major end-item; (2) reduce item demand for maintenance manpower and logistic support; (3) provide essential management information; and (4) hold down its own impact on overall program cost and schedule.

3.2.1 Software Reliability Engineering Program Requirements

Each reliability program shall include an appropriate mix of reliability engineering and accounting tasks depending on the life cycle phase. These tasks shall be selected and tailored according to the type of item (system, subsystem or unit) and for each applicable phase of the acquisition. They shall be planned, integrated and accomplished in conjunction with other design, development and manufacturing functions. The overall acquisition program shall include the resources, schedule, management structure, and controls necessary to ensure that specified reliability program tasks are satisfactorily accomplished. Figure 3-1 illustrates the insertion of software reliability prediction and estimation into the software development process. Note that the methodology actually spans the software life cycle including reliability specification and reliability assessment once the system is operational.

Tasks shall focus on the prevention, detection, and correction of reliability design deficiencies, unreliable units, and workmanship defects. Reliability engineering shall be an integral part of the design process, including design changes. The means by which reliability engineering contributes to the design, and the level of authority and constraints on this engineering discipline, shall be identified in the reliability program plan. An efficient reliability program shall stress early investment in reliability engineering tasks to avoid subsequent costs and schedule delays.

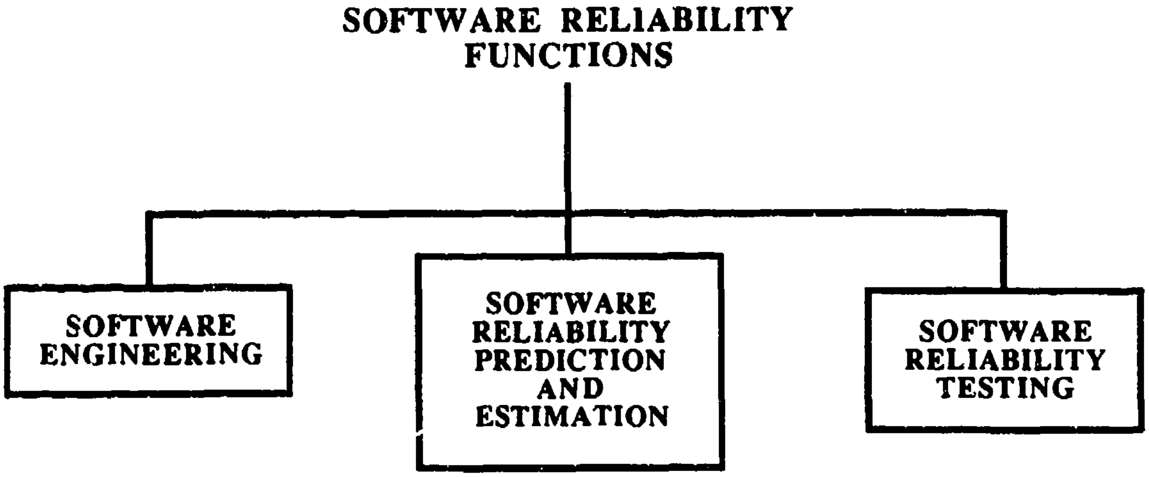
Figure 3-2 illustrates the software reliability prediction and estimation discipline in context of an overall approach to improving software reliability. As illustrated, the concerns with software reliability must permeate and be integral to the entire software development process. In fact, these same disciplines are applicable to post deployment software support, i.e., software logistics support. The developers must approach the software development with reliability as a goal. Use of formal approaches such as MIL-STD-2167A, modern techniques and tools, provide the foundation for building reliability into the product. The testing process must also account for reliability demonstration. RADC TR 84-53, Software Test Handbook, provides a methodology for planning testing techniques and tools which aid in meeting testing objectives. The prediction and estimation techniques advocated in this document provide the oversight role. Companion documents are the MIL-STD-2168, which states software QA requirements for DoD software developments; RADC TR 85-37, which establishes a methodology for quality specification and measurement; RADC TR 85-47, Impact of Hardware/Software Faults on System Reliability which establishes a new modeling approach to software reliability; and RADC TR 83-176, which is a guidebook on the use of existing software reliability models.

The incorporation of this approach in software developments promises significant benefit. This general approach could be viewed as a software reliability discipline. Functions of that discipline are portrayed in Figure 3-2.

3.2.2 Software Reliability Program Plan

A Reliability Program Plan shall be prepared and include, but not limited to the following:

a. Recognition of the Reliability Program within the development organization responsible for the development.


SOFTWARE RELIABILITY FUNCTIONS
SOFTWARE ENGINEERING
SOFTWARE RELIABILITY PREDICTION AND ESTIMATION
SOFTWARE RELIABILITY TESTING

- MIL-STD 2167A DEFENSE SYSTEM SOFTWARE DEVELOPMENT
- MODERN MANAGEMENT APPROACHES TO SOFTWARE DEVELOPMENT
- MODERN SOFTWARE DEVELOPMENT ENVIRONMENT
- MODERN SOFTWARE DEVELOPMENT TECHNIQUES

- MIL-STD-2168 (DRAFT) SOFTWARE QUALITY PROGRAM
- RADC TR 85-37 SPECIFICATION OF SOFTWARE QUALITY ATTRIBUTES
- EMPHASIS ON QA, IV & V
- SPECIFICATION OF QUALITY (RELIABILITY) GOALS
- RADC TR-85-47 IMPACT OF HARDWARE/ SOFTWARE FAULTS ON SYSTEM RELIABILITY
- RADC TR 83-176 A GUIDEBOOK FOR SOFTWARE RELIABILITY ASSESSMENT

- RADC TR 84-53 SOFTWARE TEST HANDBOOK
- EARLY INVOLVEMENT BY TESTING ORGANIZATION
- USE OF MODERN TEST TOOLS AND TECHNIQUES
- IDENTIFIED TEST OBJECTIVES

FIGURE 3-2 SOFTWARE RELIABILITY FUNCTIONS

b. Description of the software reliability requirements established for the system and their relationship with the system reliability requirements.

c. Description of how the Software Reliability Program will be conducted to meet the software reliability requirements.

d. Establishment of responsible personnel for the conduct of the Reliability Program with appropriate authority.

e. A description of the relationship of the Reliability Program with appropriate authority.

f. A schedule of the reliability prediction and estimation activities (milestones).

g. Identification of data collection requirements and procedures to support the reliability prediction and estimation activities.

h. Description of the reliability prediction and estimation procedures to be used.

i. Identification of potential or known reliability problems.

j. Procedures for recording the status of actions to resolve the problems identified.

Activities to be included in this plan, which comprise the software reliability functions shown in Figure 3-2, are identified in Figure 3-3.

3.2.2.1 Reliability Accounting

Tasks shall focus on the provision of information essential to acquisition, development, operation, and support management, including properly defined inputs for estimates of operational effectiveness and ownership cost. An efficient reliability program shall provide this information while ensuring that cost and schedule investment in efforts to obtain management data (such as demonstrations, qualification tests, and acceptance tests) is clearly visible and carefully controlled.

3.2.2.2 Reliability Program Interfaces

The contractor shall utilize reliability data and information resulting from applicable tasks in the reliability program to satisfy Post Deployment Software Support (PDSS) requirements. All reliability data and information used and provided shall be based upon, and traceable to, the outputs of the reliability program for all maintenance support and engineering activities involved in all phases of the system acquisition.

3.2.3 Software Reliability Modeling and Prediction Report

The report shall contain a summary which provides the contractor's conclusions and recommendations based upon the analysis. Contractor interpretation and comments concerning the analysis and the recommended actions for the elimination or reduction of failure risks shall be included. A design evaluation summary of major problems detected during the analysis shall be provided in the final report. A list of software or functional elements of the system omitted from the reliability models and reliability predictions shall be included with rationale for each element's exclusion.

The report shall contain a summary which provides the contractor's conclusions and recommendations based upon the analysis. Contractor interpretation and comments concerning the

CONCEPT DEVELOPMENT/ ACQUISITION INITIATION	MISSION SYSTEM/ REQUIREMENTS	SOFTWARE REQUIREMENTS	PRELIMINARY AND DETAILED SOFTWARE DESIGN	CODING AND UNIT TESTING	CSC INTEG AND TEST/ CSCI-LEVEL TESTING	SYSTEM INTEGRATION AND TESTING/ OT&E	OPERATIONS AND MAINTENANCE
• Establish Reliability Requirements • Perform High Level Trade-offs • Relate Reliability To User	• Set Reliability Goals For System • Allocate Reliability Goals To Hardware and Software • System Reliability Assessment (SDR)	• Allocate Reliability Requirements To Software • Analyze Testability of Requirement • Analyze Feasibility of Requirements • SRR	• Decompose and Budget Reliability Requirements To Software Components • Establish Design Practices To Encourage Reliable Software Design • Analyze/ Simulate Reliability Performance • Predict Software Reliability • PDR • CDR	• Build In Reliability • Establish Coding Standards To Encourage Reliable Software Production • Conduct Unit Testing/ Debugging To Remove Module Level Faults • Prototype Builds For User Feedback • Product Software Reliability	• Test To Requirements • Test Thoroughness Evaluation • Maintain Standards During Rework • Insure Test Quality • Regression Testing • Estimate Software Reliability • Problem Report Statistics • TRR • Acceptance Testing	• Hardware/ Software Error Analysis • Hardware/ Software Reliability Integration • System Test Thoroughness Evaluation • Insure Test Quality • Regression Testing • Estimate System Reliability • Test Assessment In Operational Environment	• Regression Testing • Quality Assurance • Reliability Measurement

FIGURE 3-3 SOFTWARE RELIABILITY ENGINEERING MANAGEMENT

analysis and the recommended actions for the elimination or reduction of failure risks shall be inciuded. A design evaluation summary of major problems detected during the analysis shall be provided in the final report. A list of software or functional elements of the system omitted from the reliability models and reliability predictions shall be included with rationale for each element's exclusion.

Reliability critical software components of the system extracted from the reliability modeling and reliability prediction effort shall be listed and included in the summary. Reliability critical software components include high failure rate components (experienced during testing), real-time processing components, and those components performing mission critical functions.

The data collected and results of the methods and procedures contained in this guideline should be provided as appendices to this report to substantiate the summary conclusions and the identified critical elements.

3.2.4 Software Reliability Program Tasks

The major tasks, covered by this guidebook, are described as procedures in Task Sections 100, 200, and 300 of this guidebook to do reliability predictions, test planning, and reliability estimations. Other tasks to be performed with the Software Reliability Program are described in the following paragraphs..

3.2.4.1 Monitor/Control of Subcontractors and Suppliers

The contractor shall assure that software components obtained from subcontractors or suppliers meet reliability requirements.

The contractor shall, as appropriate:

a. Incorporate quantitative software reliability requirements in subcontracted software specifications.

b. Assure that subcontractors have a Reliability Program that is compatible with the overall program and includes provisions to review and evaluate the software to be delivered.

c. Attend and participate in subcontractor's design reviews.

d. Review subcontractor's predictions and estimations for accuracy and correctness of approach.

e. Review subcontractor's test plans, procedures, and reports.

f. Require delivery of appropriate data collected in accordance with the Reliability Program.

g. Assure the subcontractors have and are complying with corrective action reporting procedures and follow-up corrective actions.

h. Monitor reliability demonstrations tests.

A reference document is MIL-STD 2168.

3.2.4.2 Program Reviews

The Reliability Program shall be planned and scheduled to permit the contractor and the Government to review program status. Formal review and assessment of contract reliability requirements shall be conducted at major program points, identified as system program reviews, as specified by the contract. As the program develops, reliability progress shall also be assessed by the use of additional reliability program reviews as necessary. The contractor shall schedule reviews as appropriate with his subcontractors and suppliers and insure that the Government is informed in advance of each review.

The reviews shall be identified and discuss all pertinent aspects of the reliability program such as the following, when applicable:

a. At the Software Requirements Review

  1. Identify reliability requirements in terms of fault density and failure rate (see Table TS100-1).

  2. Establish allocation of software reliability requirements to software components (CSCI).

b. At the Preliminary Design Review (PDR):

  1. Update reliability status including:

     (a). Reliability modeling.
     (b). Reliability apportionment.
     (c). Reliability predictions.
     (d). Failure Modes, Effects and Criticality Analysis (FMECA).
     (e). Reliability content of specification.
     (f). Design guideline criteria.
     (g). Other tasks as identified.

  2. Determine other problems affecting reliability.

  3. Review Reliability Critical Items Program.

c. At the Critical Design Review (CDR), review:

  1. Reliability content of specifications.
  2. Reliability prediction and analyses.
  3. Reliability critical items program.
  4. Other problems affecting reliability.
  5. FMECA.

d. At Interim Reliability Program Reviews, review:

  1. Discussion of those items reviewed at PDRs and CDRs.
  2. Results of failure analyses.
  3. Test schedule: start dates and completion dates.
  4. Component design, reliability and schedule problems.
  5. Status of assigned action items.
  6. Contractor assessment of reliability task effectiveness.

7. Other topics and issues as deemed appropriate by the contractor and the Government.

e. At the Test Readiness Review, review:

1. Reliability analyses status, primarily prediction.
2. Test schedule.
3. Test profile.
4. Test plan including failure definition.
5. Test report format.
6. FRACAS implementation.

A reference document is MIL-STD 1521A.

3.2.4.3 Failure Reporting, Analysis, and Corrective Action System (FRACAS)

The contractor shall have a closed loop system that collects, analyzes, and records failures that occur for specified levels of the software prior to acceptance by the procuring activity. The contractor's existing data collection, analysis and corrective action system shall be utilized, with modification only as necessary to meet the requirements specified by the Government.

Procedures for initiating failure reports, the analysis of failures, feedback of corrective action into the design, manufacturing and test processes shall be identified. The analysis of failures shall establish and categorize the cause of failure.

The closed loop system shall include provisions to assure that effective corrective actions are taken on a timely basis by a follow-up audit that reviews all open failure reports, failure analyses, and corrective action suspense dates, and the reporting of delinquencies to management. The failure cause for each failure shall be clearly stated.

When applicable, the method of establishing and recording operating time shall be clearly defined.

The contractor's closed loop failure reporting system data shall be transcribed to Government's forms only if specifically required by the procuring activity. Appendices B, C, and D provide appropriate forms. A reference document is MIL-STD 785B.

3.2.4.4 Failure Review Board (FRB)

The FRB shall review functional/performance failure data from appropriate inspections and testing including subcontractor qualification, reliability, and acceptance test failures. All failure occurrence information shall be available to the FRB. Data including a description of test conditions at time of failure, symptoms of failure, failure isolation procedures, and known or suspected causes of failure shall be examined by the FRB. Open FRB identified items shall be followed up until failure mechanisms have been satisfactorily identified and corrective action initiated. The FRB shall also maintain and disseminate the status of corrective action implementation and effectiveness. Minutes of FRB activity shall be recorded and kept on file for examination by the procuring activity during the term of the contract. Contractor FRB members shall include an appropriate representative to the FRB as an observer. If the contractor can identify and utilize an already existing and operating function for this task, then he shall describe in his proposal how that function, e.g., a Configuration Control Board (CCB), will be employed to meet the procuring activity requirements. This task shall be coordinated with Quality Assurance organizations to insure there is no duplication of effort. A reference document is MIL-STD-785B.

3.2.4.5 Critical Reliability Component Identification

Based on the Software Reliability Program, the predictions and estimations, and other analyses and tests, the contractor shall identify these software components which potentially have high risk to system reliability. Techniques such as Failure Modes, Effects and Criticality Analysis (FMECA), Sneak Circuit Analysis (SCA), Design and Code Inspections, Walk-throughs, etc. are recommended to assist in this identification process. A reference document is MIL-STD-785B.

3.2.4.6 Test Program

The Reliability Program shall be closely coordinated with the Test Program. The Test Program shall include a Reliability Qualification Test to demonstrate achievement of the reliability requirements. The Test Program shall be specified by reference to appropriate Military Standards. Reference documents are MIL-STD-781C and MIL-STD-2167A. Task Section 200 provides guidance for specific test technique planning with companion document, the Software Test Handbook, RADC TR 84-53.

3.2.5 Implementation

The reliability program shall be initiated at the early phases of a project. The program and its associated plan should be required in the acquisition planning documents and RFP.

Reliability prediction shall be initiated early in the definition stage to aid in the evaluation of the system architecture and design and to provide a basis for system reliability allocation (apportionment) and establishing corrective action priorities. Test Planning begins in the early phases of development. Guidance in this guidebook supports that planning by assisting in testing strategies and technique selection. Reliability estimation shall be initiated early in the test phases utilizing the observed failure rate during testing as a basis to estimate how the software will behave in an operational environment. Reliability predictions and estimations shall be updated when there is significant change in the system design, availability of design details, environmental requirements, stress data, failure rate data, or service use profile. A planned schedule for updates shall be specified by the procuring activity.

3.2.6 Ground Rules and Assumptions

The Government Program Office or contractor shall develop ground rules and analysis assumptions. The ground rules shall identify the reliability prediction and estimation approach in terms of this Guidebook, the lowest indenture level to be analyzed, and include a definition of mission success in terms of performance criteria and allowable limits. The SPO or contractor shall develop general statements of item mission success in terms of performance and allowable limits for each specified output. Ground rules and analysis assumptions are not inflexible and may be added, modified, or deleted if requirements change. Ground rules and analysis assumptions shall be documented and included in the reliability prediction and estimation report.

3.2.7 Indenture Level

The indenture level applies to the software or functional level at which the software configuration is defined. Unless otherwise specified, the contractor shall establish the lowest indenture level of analysis using the following guidelines:

a. The level specified for the prediction measurement to ensure consistency and allow cross referencing.

b. The specified or intended maintenance level for the software.

The methodology described in this guidebook supports reliability prediction and estimation at the system, CSCI, CSC, and unit levels.

3.2.8 Coding System

For consistent identification of system functions and software elements, the contractor shall adhere to a coding system based upon the software breakdown structure, work unit code numbering system of MIL-STD-780, or other similar uniform numbering system. The coding system shall be consistent with the functional block diagram numbering system to provide complete visibility of each modeled element and its relationship to the item.

3.2.9 Coordination of Effort

Reliability and other organizational elements shall make coincident use of the reliability predictions and estimations. Considerations shall be given to the requirements to perform and use the reliability predictions and estimations in support of a reliability program in accordance with MIL-STD-785B, maintainability program in accordance with MIL-STD-470, safety program in accordance with MIL-STD-882, survivability and vulnerability program in accordance with MIL-STD-2072, logistics support analysis in accordance with MIL-STD-1388, maintenance plan analysis (MPS) in accordance with MIL-STD-2080, fault diagnosis analysis in general accordance with MIL-STD-1591, and other contractual provisions.

3.3 Procedures

This guidebook has been organized to facilitate the user performing the tasks associated with reliability prediction, test planning, and reliability estimation. The tasks have been documented as procedures and are organized in Task Sections 100, 200, and 300 respectively.

3.3.1 Goal Specification

An initial step of the reliability engineering program is determining or specifying the software reliability goals for the system. The software reliability goals must be stated in the context of the system.

3.3.1.1 Mission Reliability Definition

System reliability for mission is assumed to be represented by a series arrangement of hardware, software, and possibly other components as shown in Figure 3-4. The mathematical formulation for the system mission reliability is therefore

$$R=RH*RS*RX$$

Hardware-software interactions, such as software failures induced by hardware anomalies, or failures of hardware reconfiguration caused by software faults, must be included in the RX term. Other components that may have to be added to the series model include the personnel subsystem and support equipment (power, air conditioning, etc.). Only the prediction or estimation of the RS component is covered by this Guidebook.

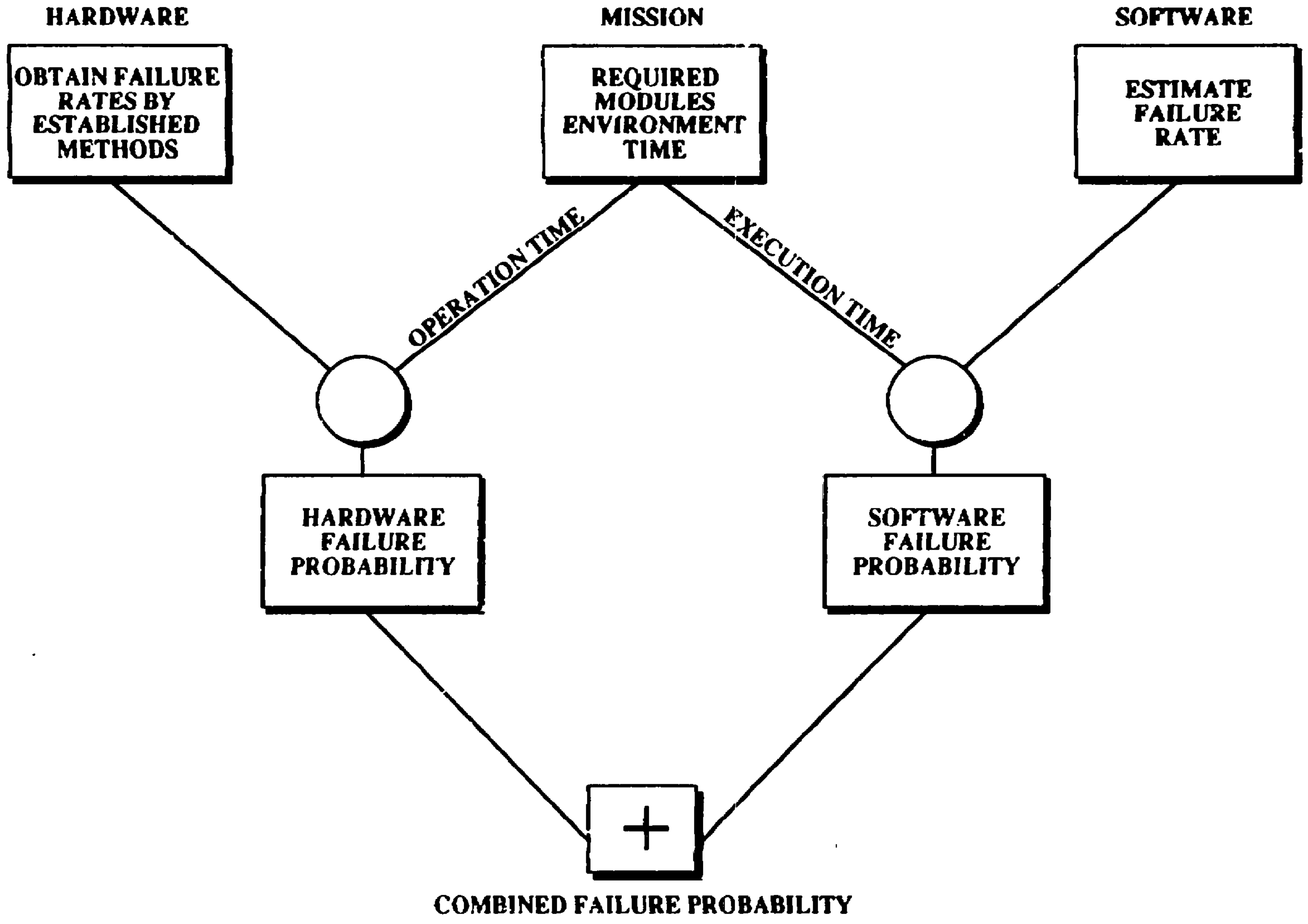

HARDWARE
MISSION
SOFTWARE
OBTAIN FAILURE RATES BY ESTABLISHED METHODS
REQUIRED MODULES ENVIRONMENT TIME
ESTIMATE FAILURE RATE
OPERATION TIME
EXECUTION TIME
HARDWARE FAILURE PROBABILITY
SOFTWARE FAILURE PROBABILITY
+
COMBINED FAILURE PROBABILITY

FIGURE 3-4 RELATIONSHIP BETWEEN HARDWARE AND SOFTWARE RELIABILITY

If the reliability of individual components is high, eg., at least 0.95, a good approximation of the system reliability can be obtained by

$$F=FH+FS+FX$$

where all F terms are mission failure probabilities (R=1-F). The software mission failure probability is the product of the software failure rate and the mission duration, expressed in identical units of time.

Where mission phases differ in hardware or software utilization or environment, a separate reliability model is required for each phase, and the total mission reliability is the series combination (product) of the individual mission phases. Differences in software utilization are presented if (a) functionally distinct software is utilized, such as automatic approach and landing software in an aircraft flight control system, or (b) there is a substantial difference in the mix of software functions. Differences in the software environment are present if there are substantial changes in the computer workload.

3.3.1.2 Quantitative Software Reliability Requirements

The software system reliability requirements shall be specified contractually.

There are three different categories of quantitative reliability requirements: (1) operational requirements for applicable software reliability parameters; (2) basic reliability requirements for software design and quality; and (3) statistical confidence/decision risk criteria for specific reliability tests. These categories must be carefully delineated, and related to each other by clearly defined audit trails, to establish clear lines of responsibility and accountability.

Software reliability parameters shall be defined in units of measurement directly related to operational readiness, mission success, demand for maintenance manpower, and demand for maintenance support, as applicable to the type of system. Operational requirements for each of these parameters shall include the combined effects of design, quality, operation, maintenance and repair in the operational environment. The basic measurement used in this guidebook for software reliability is failure rate. Definitions are provided in Appendix A.

Software reliability requirements can be incorporated in a Request for Proposal (RFP) or specification at several levels of detail. Example approaches to specifying software reliability requirements are identified in this paragraph. The various approaches depend on the level of specificity of the system and the formality of the review and evaluation process to be incorporated in the contract.

At a minimum, software reliability should be identified as a goal of the project. This can be done by stating it as a goal and providing a definition (see Appendix A). It is also beneficial to describe why reliability is important, i.e., put it in the context of the system. For example, software reliability might be particularly important in the real time message processing subsystem of a command and control system.

At another level of detail, reliability can be specified by identifying the attributes desired in the software. These attributes will then by the subject of review during the development process. RADC TR 85-37, Specification of Software Quality Attributes, or Task Section 100 of this guidebook can be used to identify the attributes and how they will be measured. This concept can be built into the source selection process also by requiring in the RFP that bidders describe their approach to providing these attributes in the software they are to develop. During the contract, such documents as the Software Development Plan, the Software Quality Assurance Plan, and the design documents would be reviewed for these attributes.

At a third level, specific quantitative reliability goals can be specified. Industry averages for application areas are described in Task Se.:tion 100 of this Guidebook. Acceptance Tests can be required to demonstrate required performance. Task Section 100 and 300 provide techniques for "measuring" the progress toward achieving the required performance and therefore can be established as review mechanisms within the development process. The RFP should identify specifically the Task Sections to be applied and the reviews where results are to be reported.

3.3.2 Prediction and Estimaticn Procedures

The steps set forth in this paragraph define the general procedures for developing a software reliability model and performing a reliability prediction or estimation. Specific tasks are contained in the Task Sections 100 and 300 of this guidebook. Figure 3-5 provides a road map for use of the procedures and tasks. The effort to develop or collect the information for the steps shall be closely coordinated with related program activities (such as design engineering, system engineering, Quality Assurance, and logistics) to minimize duplications and to assure consistency and correctness.

The steps are:

a. Define the software component level for prediction (see paragraph 3.3.2.2).

b. Identify life cycle and prediction and estimation milestones (see paragraph 3.3.2.3).

c. Identify tasks and data collection procedures.

d. Obtain or develop system architecture diagram to appropriate component leve (requires allocation of software component to hardware components) (see reliability prediction Task Section 100).

e. Define software components (see Task Section 100).

f. Define reliability model (see Task Section 100).

g. Implement tasks and data collection procedures.

h. Proceed through Prediction Procedures (see individual Reliability Prediction Tasks 101 through 104).

i. Proceed through Estimation Procedures (See individual Reliability Estimation Tasks 301 through 302).

3.3.2.1 Comparison with Hardware Reliability Prediction

Reliability prediction for hardware is an established technique, and it is therefore useful to compare the proposed software reliability procedures with those in use in the hardware field. The governing documents for hardware reliability prediction for DoD applications is MIL-STD-756B "Reliability Modeling and Prediction", and MIL-STD-785B, "Reliability Program for Systems and Equipment Development and Production". The essential steps for reliability prediction identified in MIL-STD-756B have parallel equivalent procedures for software with one exception. That exception is the absence of software equivalents for step e. Hardware components consist of separate parts, each of which may be used in many other applications, such as a 1A 250V diode or a 16K dynamic RAM chip. Failure rates can be established for these parts either from test or from analysis of field data. The procedures of MIL-STD-756B assume that the reliability of a

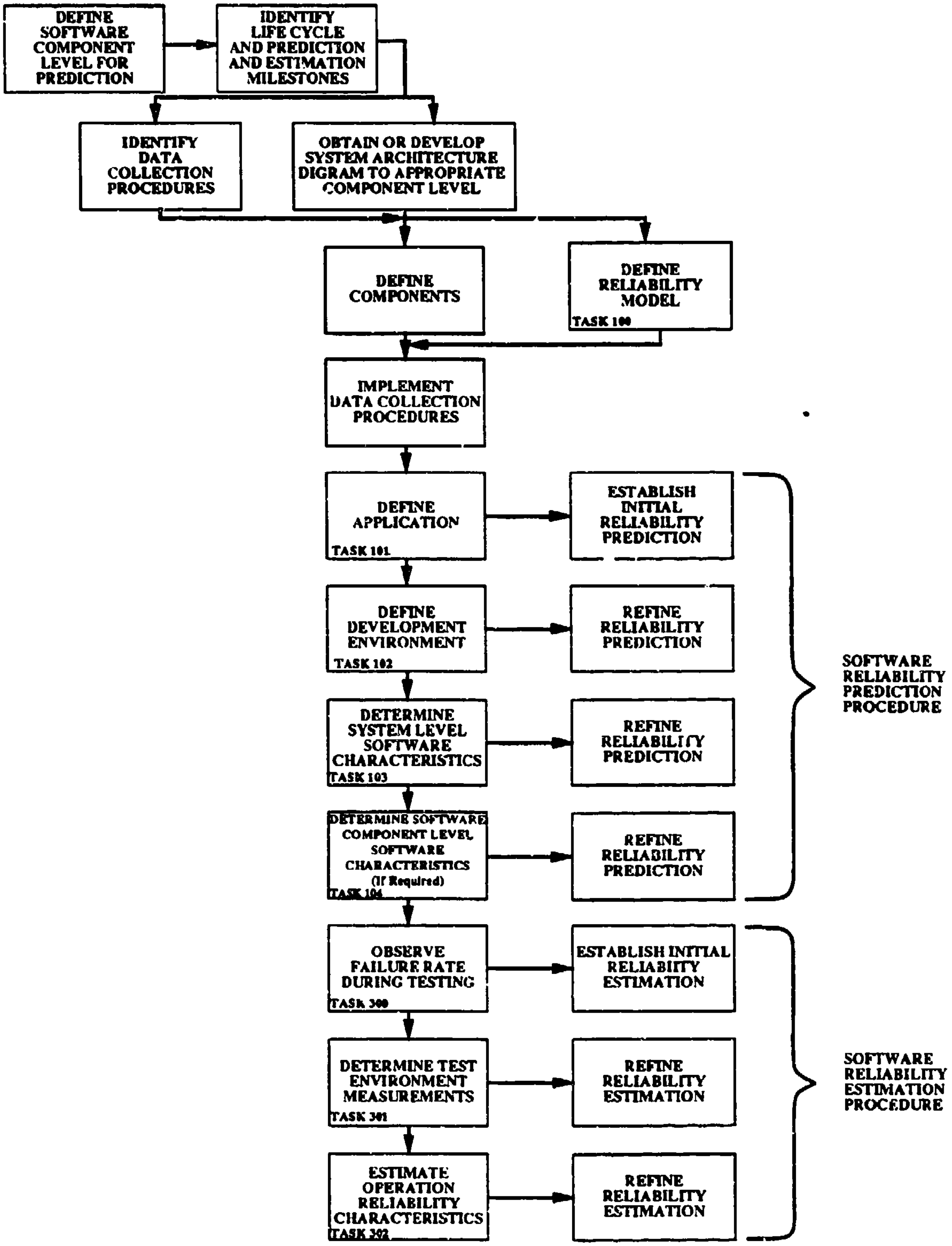

DEFINE SOFTWARE COMPONENT LEVEL FOR PREDICTION
IDENTIFY LIFE CYCLE AND PREDICTION AND ESTIMATION MILESTONES
IDENTIFY DATA COLLECTION PROCEDURES
OBTAIN OR DEVELOP SYSTEM ARCHITECTURE DIGRAM TO APPROPRIATE COMPONENT LEVEL
DEFINE COMPONENTS
DEFINE RELIABILITY MODEL
TASK 100
IMPLEMENT DATA COLLECTION PROCEDURES
DEFINE APPLICATION
TASK 101
ESTABLISH INITIAL RELIABILITY PREDICTION
DEFINE DEVELOPMENT ENVIRONMENT
TASK 102
REFINE RELIABILITY PREDICTION
DETERMINE SYSTEM LEVEL SOFTWARE CHARACTERISTICS
TASK 103
REFINE RELIABILITY PREDICTION
DETERMINE SOFTWARE COMPONENT LEVEL SOFTWARE CHARACTERISTICS (If Required)
TASK 104
REFINE RELIABILITY PREDICTION
SOFTWARE RELIABILITY PREDICTION PROCEDURE
OBSERVE FAILURE RATE DURING TESTING
TASK 300
ESTABLISH INITIAL RELIABIITY ESTIMATION
DETERMINE TEST ENVIRONMENT MEASUREMENTS
TASK 301
REFINE RELIABILITY ESTIMATION
ESTIMATE OPERATION RELIABILITY CHARACTERISTICS
TASK 302
REFINE RELIABILITY ESTIMATION
SOFTWARE RELIABILITY ESTIMATION PROCEDURE

FIGURE 3-5 SOFTWARE RELIABILITY PREDICTION AND ESTIMATION PROCEDURES

component is the product of the reliability of its (series connected) parts. The software analog to this would be to test individual assignment, branching, and I/O statements and to declare the reliability of a procedure to be the product of the reliability of its individual statements. This analog is faulty because: (a) statements cannot be meaningfully tested in isolation and (b) many software failures arise not from faults in a single statement, but rather from interactions between multiple statements (or from interactions between hardware and software). As reusable software gains wider acceptance, the assignment of a reliability index (equivalent to parts failure rate) to standard procedures may become practicable but this is still in the future.

The application of the other steps to software reliability prediction is discussed below.

The following paragraphs describe the application to software reliability prediction of those steps of the MIL-STD-756B procedure that were found to be suitable. The lower case alphabetical designation from MIL-STD-756B is indicated in parentheses for ease of reference. Only asterisked steps are required for the prediction of fault density. These steps have been taken into account in Figure 3-5.

a.* **Define the software components to be covered by the prediction.** This includes an unambiguous identification of the component, a statement of the performance requirements and the hardware environment, and a listing of inputs and outputs by type and range. This information may be available initially only at a high level of abstraction but should be decomposed for permitting tracing predictions during successive stages of development, and comparing predictions with estimates and measurements during later periods.

b. **Define the life cycle stages to be covered by the prediction and the extent of use during each stage.** It is recognized that the failure rate of software is a function of the life cycle stage. Particularly, there are significant differences in the failure rate between test and operations, and between initial operation and mature operation. Therefore, the life cycle stage(s) for which the prediction is to be made must be identified. The probability of fault removal depends on the extent to which the software is exercised. Therefore the use (in CPU-hours) between the time the prediction is made and the target for the prediction must be known.

c. **Define the execution dependencies within the software component.** This will in general require review of a top level flow chart or block diagram of the software component in order to identify units (a unit in this context is a software element at or above the module level) that are executed:

1. Routinely -- during every invocation of the software component, or once during each defined cycle for iterative programs (e.g., closed loop control);

2. Irregularly -- segments that deal with non-routine events within the program, including exceptions to conditions postulated within the program (but not exception states of computer or operating system);

3. Conditionally -- segments that are executed only if some other (non-routine) segment had been invoked (examples are message logging or creations of new files);

4. For exception handling -- response of the program to exception states identified by the computer or operating system;

5. On demand -- segments accessed only by specific operator actions such as initialization, data base cleanup, or re-hosting.

   Discussion of a technique to represent execution dependencies is found in RADC TR 85-47.

Since both the probability of execution and the accumulated execution time will differ between these classifications, seperate reliability predictions will usually be required.

d. **Define mathematical models for the software components.** The mathematical models will represent:

   1. The predicted fault density of each segment as derived in the next section;

   2. The execution time of each segment prior to the prediction interval -to determine the expected fault removal; and

   3. The execution time of each segment during the prediction interval -to determine the failure probability.

   4. Where the prediction interval covers more than one life cycle phase (such as test and operation) a separate mathematical model will be required for each phase.

e.* **Define and describe the parts of the item.** Use application area factor. As discussed in the preceding section, a major divergence of software from hardware reliability prediction practices is due to lack of an equivalent to the hardware part. However, software reliability prediction is still based on concepts of quantity, the average number (fraction) of faults per line of code. The number of faults in a software component is thus assumed to be proportional to the number of lines of code. Although we cannot, at the present state of knowledge, identify one computer program as being made up of high failure rate parts and another one of low failure rate parts, there is evidence that high and low failure fault densities are associated with certain application areas. The application area factor captures this experience as a basic predictor of the fault density.

f. **Define the operational environment.** The operational environment determines the rate at which the faults inherent in the software will be transformed to failures. Operational environment in this sense means the environment in which the software will be operating during the interval for which the reliability prediction is to be made. It can apply to test, operation in a prototype environment, or a full scale operational environment. The most important characteristics of the operational environment which affect the reliability are:

   1. Computer performance (throughput),
   2. Variability of Data and Control States, and
   3. Workload.

   The contribution of each of these to the reliability estimation is discussed in Task Section 300.

g.* **Account for software development environment and software implementation.** Differences in the software development environment and in the software implementation affect the fault density in a manner similar to that in which stress levels affect the failure probability of parts.

h. **Define the failure distribution in execution time.** Software fails only when it is being executed. Therefore, the natural normalization factor for software failures is execution time. The software failure rate based on Computer Operation hour is analogous to the hardware failure rate ("lambda") per hour (implying operating hour of the component).

i.* **Compute the Reliability.** The algorithms for predicting fault density are discussed in Task Section 100 as well as the conversion of fault density into failure rate. The estimation of reliability based on testing experience is also described in Task Section 300.

3.3.2.2 Software Component Level

The initial step in following the prediction and estimation procedures is the determination of the level at which the software reliability will be modeled. The levels of software are defined by MIL-STD 2167A as System, Computer Software Configuration Item (CSCI), Computer System Component (CSC), and Unit. The Reliability prediction and estimation procedures on this Guidebook can be used at any of these levels.

The following procedure is recommended.

a. During early phases of development (Concept Development, Mission System/Software Definition, Software Requirements) model at software system level.

b. During design phases, model at CSCI level.

c. During coding model at the CSC level. For critical software the contracting agency may direct modeling at a lower level (such as unit). Support software or commercial off-the-shelf programs should be modeled at the CSCI level or system level.

d. During testing, model at CSCI level or, if directed, a lower level.

3.3.2.3 Identify Life Cycle

The software life cycle according to MIL-STD 2167A is illustrated in Figure 3-1. Applicable points during this life cycle when a reliability prediction or estimation is recommended are:

a. During Concept Development to support Feasibility Studies.

b. During Mission/System/Software Definition to support high level architectural studies/trade-off studies and to establish development goals/specifications. Results should be reported formally at SDR.

c. During proposal preparation by contractors for evaluation purposes.

d. During Software Requirements Analysis to support feasibility analyses. Results should be reported formally at SRR.

e. During Preliminary Design to support software architecture decision and allocation. Results should be reported formally at PDR.

f. During Detailed Design to support detailed design decisions/trade-off stuaies/algorithm development. Results should be reported formally at CDR.

g. During coding and unit testing to support developer's decision to release software to formal testing. Results can be reported through QA audit reports or problem reporting process.

h. During testing phases to support test and evaluation process and acceptance. Results can be reported at the end of each phase of testing or periodically during testing. Results of any acceptance testing should be formally reported.

i. During OT&E as formal evaluation process.

j. During post deployment support as an assessment of actual reliability achieved and to support a Reliability Improvement Program.

3.3.2.4 Limitations of Reliability Predictions

The art of predicting the reliability of software has practical limitations such as those depending on data gathering and technique complexity. Considerable effort is required to generate sufficient data to report a statistically valid reliability figure for a class of software. Casual data gathering accumulates data more slowly than the advance of technology; consequently, a valid level of data is never attained. In the case of software, the number of people participating in data gathering all over the industry is rather large with consequent varying methods and conditions which prevent exact coordination and correlation. Also operational software reliability data is difficult to examine due to the lack of suitable data being acquired. Thus, it can be seen that derivation of failure rates (being mean values) is empirically difficult and obtaining valid confidence values is practically precluded because of lack of correlation.

The use of failure rate data, obtained from field use of past systems, is applicable on future concepts depending on the degree of similarity existing both in the software design and in the anticipated environments. Data obtained on a system used in one environment may not be applicable to use in a different environment, especially if the new environment substantially exceeds the design capabilities. Other variants that can affect the stated failure rate of a given system are: different uses, different operators, different maintenance practices, different measurement techniques or definitions of failure. When considering the comparison between similar but unlike systems, the possible variations are obviously even greater.

Thus, a fundamental limitation on reliability prediction is the ability to accumulate data of known validity for the new applications. Another fundamental limitation is the complexity of prediction techniques. Very simple techniques omit a great deal of distinguishing detail and the prediction suffers inaccuracy. More detailed techniques can become so bogged down in detail that the prediction becomes costly and may actually lag the principal development effort.

This Guidebook includes two methods: reliability prediction and reliability estimation. These methods vary in degree of information needed and timing of their application. References to other or complementary methods are provided.

The content of this Guidebook has not been approved by the Military Services and has not been coordinated with appropriate segments of industry. It provides an initial attempt to document a methodology that would provide a common basis for reliability predictions during acquisition programs for military systems. It also establishes a common basis for comparing and evaluating reliability predictions of related or competitive designs. The failure rates and their associated adjustment factors presented herein are based upon evaluation and analysis of the best available data at the time of issue.

3.3.3 Test Technique Selection Procedure

Integral to a Reliability Program is an effective and efficient Test Program (see paragraph 3.2.4.6). The Test Program conducted should be in accordance with the contract requirements and applicable Military Standards (e.g., MIL-STD-2167A). The procedures documented in Task Section 200 update the procedures established in the Software Test Handbook, RADC TR 84-53. They have been updated based on further data collection and experimental analyses. The procedures assist in test technique selection for more effective and comprehensive testing.

The phases of testing covered by these procedures is shown in Figure 3-1.

TASK SECTION 100

SOFTWARE RELIABILITY PREDICTION

TASK SECTION 100
SOFTWARE RELIABILITY PREDICTION
OVERVIEW OF METHODOLOGY

100.1 Purpose

The purpose of task 100 is to describe the general procedures for predicting software reliability, in terms of Fault Density (FD) based on the characteristics of the Application type, the Development Environment, and the Software Implementation.

100.2 Documents Referenced in Task Section 100

MIL-STD 2167A

MIL-STD 2168

RADC TR 85-37

RADC TR 85-47

MIL-STD 756B

MIL HDBK 217D

MIL-STD 785B

RADC TR 87-171

100.3 General Procedures

Make the Reliability Prediction. Use the measurements in Tasks 101, 102, 103 and 104 to predict reliability, in terms of a Reliability Prediction Figure of Merit (RPFOM), as follows:

a. Project Initiation:
   Task 101 - Use metric A as prediction: RPFOM = A.
   Task 102 - Use metrics A, D: RPFOM = A * D.

b. Requirements and Design Phases:
   Task 103 - Use metrics A, D, S1: RPFOM = A*D*S1.

c. Coding Phase:
   Task 104 - Use metrics A, D, S1, S2: RPFOM = A*D*S1*S2

100.3.1 System Architecture

A system architecture diagram should be obtained or developed. This diagram should show a high level allocation of software components (typically at the CSCI level) to hardware components. If available, control flow and or data flow diagrams prepared by the design team are valuable for preparation of the reliability model.

100.3.2 Definition of Components

Each software component to be modeled should be identified and defined. This information is typic.illy available in a system/subsystem specification. See paragraph 3.3.2.2 for a description of suitable component levels.

100.3.3 Reliability Model

Based on the system architecture diagram, the software components allocated to hardware components can be identified. This allocation should be overlayed on the hardware reliability block diagram. The reliability block diagram shows interdependencies among all elements or functional groups of the system. The purpose of the reliability block diagram is to show by concise visual shorthand the various series - parallel block combinations (paths) that result in successful mission performance. A complete understanding of the system's mission definition and service use profile (operational concept) is required to produce the reliability diagram.

At this point, two approaches can be taken. The first is to utilize the prediction techniques described in Tasks 101 through 104 to calculate a Reliability Prediction Figure of Merit (RPFOM) for each component identified in the block diagram. This is typically at a CSCI level. The second approach is to model at a lower level the software processing within each software component.

100.3.3.1 Reliability Model 1

For each software component or component grouping on the block diagram, follow Tasks 101 through 104. These tasks provide the procedures for calculating a predictive Reliability Prediction Figure of Merit (RPFOM) according to the following equation:

$$RPFOM = A * D * S$$

where RPFOM is the predicted fault density, A the application type metric, D the software development environment metric, and S the software characteristic metric. A is expressed in (fractional) faults per line of code, and examples of actual values are presented in Task 101. D and S are modification factors, and each of these can have a value that is less than one (1) if the environment or implementation tends to reduce the fault density, or a value of greater than one if it tends to increase fault density. These factors are equivalent to pi factors in MIL HDBK 217E. The Application Area metric (A) represents an industry average or baseline fault density which can be used as a starting point for the prediction if information for determining D or S is missing. The Tasks 101 through 104 are procedures for prediction. The tables, coefficients, and algorithms will be updated as a result of data collection and statistical analyses being performed on more software systems on a continuing basis. Refer to the data collection procedures and worksheets referred to in the body of these tasks, and contained in Appendix B, to comply with this prediction methodology. This is a generic process and should be applicable to all software components.

100.3.3.2 Reliability Model 2

For specified software components, a detailed model based on a functional flow analysis can be developed. A functional decomposition of the software component is required as well as a mission thread analysis. For each subcomponent as defined by the procuring authority, the procedures described in 100.3.3.1 (Reliability Model 1) can be used to devise an RPFOM. The flow between these subcomponents with individual reliability numbers can be modeled as a Markov Process. RADC TR 85-47 describes this modeling approach.

TABLE TS100-1
BASELINE FAULT DENSITY AND FAILURE RATE VALUES

	FAULT DENSITY			FAILURE RATE	
APPLICATION	NO. OF SYSTEMS	TOTAL LINES TOTAL LINES OF CODE	AVG FAULT DENSITY BY SYSTEM	NO. OF SYSTEMS	OPTATIONAL FAILURE RATE
AIRBORNE	7	540,617	.013	1*	.08*
STRATEGIC	21	1,793,831	.009	5	0.0109
TACTICAL	5	88,252	.008	5	0.108
PROCESS CONTROL	2	140,090	.002	-	-
PRODUCTION CENTER	12	2,575,427	.009	4	0.198
DEVELOPMENTAL	6	193,435	.014	3*	11.8*
TOTAL/AVERAGE	53	5,331,652	.010	14	0.1

•These two values are failure rates during test, not included in total

100.3.3.3 Software Reliability Prediction

The results of using Reliability Model 1 or 2 is a prediction of software reliability for each block in the system/hardware block diagram. A description of the format and documentation required for a block diagram is in MIL-STD 756B, Task Section 100. The software reliability prediction numbers should be entered on the block diagram and incorporated into the mathematical model of that diagram. The use of these procedures and assumptions made should be documented under paragraph 2.3.8.1, Software Reliability Assumptions in that task section.

When using Model 1, the predicted software reliability figure of merit is a fault density as described above. When using Model 2, the predicted software reliability figure of merit is a probability that the software will not cause of failure of a mission for a specified time under specified conditions. The probability of failure for a specified period of time is given by the failure rate, the expected (average) number of failures per unit time, usually taken as a computer- or CPU-hour. Because the failure rate has a direct correspondence to the definition of software reliability, it is selected as the primary unit of measure for software reliability.

The fault density, predicted by Model 1 is used as an early indicator of software reliability based on the facts that: (1) the number of problems being identified and an estimate of size are relatively easy to determine during the early phases of a development and (2) most historical data available for software systems support the calculation of a fault density, but not failure rate. Fault density is the number of faults detected (or expected to be detected) in a program divided by the number of executable lines. Fault density was found to average from .01 to .02 in high quality software, in early research on software reliability (1970-1980). The prediction of fault density does not require knowledge of the execution environment, and thus it is suitable for the early stages of software development. As information about the intended execution environment becomes available, the predicted fault density can be translated into a predicted failure rate.

The fault density cannot be used directly in the system block model. Instead it can be used as an indicator for unreliable components or critical reliability components. The fault density derived by the prediction methods can be compared to Table TS100-1 which contains industry averages or with the specified fault density requirement, if stated in the RFP. Actions can then be taken in the early phases of development to remedy pinpointed unreliable components through redesign, reimplementation or emphasis and rework during test.

A transformation mechanism between fault density and failure rate is based on the following. A faulty statement will not result in a failure under any circumstances until it is executed, i.e., until it affects either the memory content or the control state of a computer. Given that a fault exists, the probability of initiating a failure is dependent on three characteristics of the execution environment:

a. Computer performance (throughput),

b. Variability of data and control states, and

c. Workload.

These characteristics affect both test and operation and the metrics applied to them are discussed in Task Section 300.

The following three approaches can be used for the transformation:

a. Using established empirical values, such as are included in Table TS100-2.

b. Developing a theoretically based transformation function.

c. Using in-house data to derive an empirical relationship.

As a baseline for the transformations discussed here, Table TS100-1 provides currently available data. Using the Average row, a transformation ratio of .1/.0094 = 10.6, operational failure rate to fault density. Examination of Table TS100-1 shows that for individual application categories contributing to that average, the transformation ranges from 1.2 to 23. Table TS100-2 is provided as currently available transformation for the individual application areas.

The second approach requires the following deviation and data collection. Practically all software failure rate models postulate a direct functional relationship between the fault content of a program and its failure rate. In the simplest case, the functional relation is a constant, e.g., the failure (hazard) rate is proportional to the expected value of the number of faults remaining.

These relations permit the estimation of fault content, given the failure rate, or vice versa. Two cases are the estimation of the number of faults removed in a give time interval (expressed in terms of execution time). For the first of these we use:

$$L - L_0 * \exp(-Qt/No)$$

where L is the failure rate at time t, Lo is the initial failure rate, Qt is a factor that is considered constant in a given environment, and No is the initial fault content. Given the program size, the fault content can be converted to fault density.

The number of faults removed during a time interval can be obtained from:

$$n = Q1 * (L1 - L2)$$

where n is the fault decrement, Q1 a constant in a given environment and L1 and L2 are the failure rates at the beginning and end of the period over which the fault removal is estimated.

In spite of the mathematical simplicity of these formulas, considerable effort is usually required to find values for the constants Qt and Q1 that are applicable to generic environments.

The third approach requires that failure rate data be collected during operation of the software and compared with the fault density recorded during the development. This is possible if parts of the system are implemented prior to other parts, i.e., an incremental development and those parts that are implemented early are put through an IOT&E phase of testing. Another situation where data may be available is in an environment where a new system or a new generation of an old system is being developed and existing fault density and failure rate data has been collected on the existing system and can be compared with the new development. Data Collection Procedures 5, 6, and 7 (Appendix B) can be used to calculate fault density in Tasks 103 and 104.

If one of the empirical approaches (first and third approach) is used, the computer throughput must be taken into account if the baseline is derived from a different computer than the target for the intended application. Computer performance determines the frequency with which statements are executed. All other things being equal, a program continuously executing on a fast computer will experience a higher failure rate than the same program executing on a slower computer.

Failure rates expressed in computer-hours (also referred to as wall-clock-hours) or CPU-hours are the most useful reliability metrics in a given environment, but it must be recognized that the failure exposure of a program is dependent on the number of executions rather than on passage of time. Thus, if one pass through a program with a given data set takes 1 second on computer A and 0.1 second on computer B, then the failure exposure per unit time imparted by the latter is ten times

TABLE TS100-2
TRANSFORMATION FOR
FAULT DENSITY TO FAILURE RATE

APPLICATION TYPE	TRANSFORMATION RATIO
AIRBORNE	6.28
STRATEGIC	1.2
TACTICAL	13.8
PROCESS CONTROL	3.8
PRODUCTION CENTER	23
DEVELOPMENTAL	132.6*
AVERAGE	10.6

* These data represent a transformation ratio derived from failure rates observed during testing, not operation.

that of the former. Other things being equal, one expects the failure rate (expressed in common time units) in B to be ten times that of running the program in A.

A customary measure of computer performance is the instruction processing rate expressed as MIPS (Million Instructions Per Second). Although this relates to the native instruction set of each computer, and is therefore not strictly comparable across computer types, it can form a working basis for most of the transformations required for reliability prediction.

A faulty program executing on a computer, even on a very fast computer, will not experience software failures if it constantly operates on a data set that has already been run correctly. On the other hand, introducing deliberate variability into the input data, as in a test environment, will accelerate the occurrence of failures. Thus, metrics for capturing the variability of the environment are an important component of the transformation procedures.

The workload of the computer system affects the software failure rate even if the execution frequency of a given program is held constant (e.g., in a multi-tasking environment where the workload is a composite of several programs.) It has been found that at very high workloads, the failure rate can increase by more than an order of magnitude over the baseline (low workload) rate. Suitable metrics are discussed in later sections.

The primary use of the transformation mechanism is to permit reliability prediction using fault density level to be translated to failure rate prediction.

100.4 Detail to be Specified by the Procuring Authority.

a. Requirement of Tasks 101 through 104.

b. Statement of reliability requirements.

c. Define the software component level for prediction (different levels may be specified for each life cycle phase).

d. Define life cycle phases to be covered and prediction milestones.

e. Identify data collection procedures in Tasks 101 through 104.

f. Identify fault density/failure rate transformation procedure to be used.

100.5 Overview of the Process

RPFOM data collection procedures (Appendix B) should be applied to software projects during their development. These procedures also may be applied to completed projects in a manner which emulates their development. Only project data sources which (would have) exist(ed) at the software life-cycle phase corresponding to the metrics of interest are referenced for each RPFOM calculation. This careful attention to the timeliness of all data sources is necessary in order to meaningfully apply the reliability prediction methodology.

The RPFOM data to be collected for a project is specified in metric worksheets (Appendix B). Each worksheet targets a specific metric(s) (e.g., Quality Review), software life-cycle phase (e.g., Detailed Design), and software component level (i.e., System, CSCI, or Unit) as illustrated in Table TS100-3. Table TS100-4 lists all applicable data sources for each of the data collection forms. The collected data can be recorded manually on metric answer sheets prior to entry into an automated RPFOM Database. This will facilitate data entry and reduce the impact of multiple users accessing a single workstation. Exceptions are noted in Tasks 101 through 104. Each answer

TABLE TS100-3 FRAMEWORK FOR APPLICATION OF METRIC WORKSHEETS (Designated by "W/S")

	LIFE-CYCLE PHASE	SOFTWARE PRE-DEVELOPMENT		SOFTWARE REQUIREMENTS ANALYSIS	PRELIMINARY DESIGN	DETAILED DESIGN	CODING AND UNIT TESTING	
	REVIEW	SRR		SSR	PDR	CDR		
Application Level	RPFOM Source Documents	A	A x D	A x D x S1 (S1 = SA x ST x SQ)			A x D x S1 x S2 (S2 = SL x SM x SX x SR)	METRIC (ACRONYM)
SYSTEM	SSS	W/S 0						APPLICATION TYPE (A)
	SDP		W/S 1A or 1B					DEVELOPMENT ENVIRONMENT (D)
CSCI	SRS			W/S 2A				ANOMALY MANAGEMENT (SA)
	STLDD				W/S 2B			
	SDDD					W/S 2D		
UNIT	SDDD					W/S 2C		
CSCI	SRS			W/S 3A				TRACEABILITY (ST)
	STLDD				W/S 3B			
	SDDD					W/S 3C		
	SRS			W/S 4A				QUALITY REVIEW (SQ)
	STLDD				W/S 4B			
	SDDD					W/S 4D		
UNIT	SDDD					W/S 4C		
	CODE						W/S 8D	LANGUAGE TYPE (SL) MODULARITY (SM) COMPLEXITY (SX)
CSCI	CODE						W/S 9D 10D	LANGUAGE TYPE (SL) MODULARITY (SM) COMPLEXITY (SX)
UNIT	CODE						W/S 11D	STANDARDS REVIEW (SR)
CSCI	CODE							

TABLE TS100-4 RPFOM DATA REQUIREMENTS

Task Section	Data	Other Input	Data Collection Worksheet	Data Collection Procedure
101	Application (A)	System architecture diagram; statement of need; required operational capability; system requirements statement	0	0
102	Development Environmant (DE)	Requirements document; Specifications document	1	1
103	System Level Characteristics (S)	S1		
	Requirements and Design Representation Metric (S1)	SA x ST x SQ		
	Anomaly Management (SA)	All system documentation and source code	2	2
	Traceability (ST)	Requirements and design documents with a cross-reference matrix	3	3
	Quality Review Results (SQ)	Requirements document; preliminary design specification; detailed design specification	4	4
	Discrepancies (DR)	Discrepancy reports	6	6
104	Software Implementation Metric (S2)	SL x SM x SX x SR		
	Language Type (SL)	Requirements specification	8	6 and 8
	Modularity (SM)	Module size estimates and source code	9D	9
	Complexity (SX)	Source code	10D	10
	Standards Review (SR)	Source code	11D	11

sheet (Appendix C) supports all worksheets corresponding to a specific software component level and life-cycle phase.

Data collection and computational requirements for each RPFOM are summarized in the following sections. Figure TS100-1 summarizes the RPFOM data collection. Reference Tasks 101 through 104 for details.

100.5.1 Summary of Computations

The goal of software reliability prediction is the assessment of a software system's ability to meet specified reliability requirements. The RPFOM is a statement of predicted reliability based on software measurements collected during each phase of software development prior to Enter Figure commencement of CSC integration and testing. The RPFOM contributes to attainment of reliability goals by providing feedback for improving software system design and implementation.

The functional definition of RPFOM is that described in the SRPEG for Reliability Model 1:

$$\text{RPFOM} = \text{Predictive Fault Density} = \text{Faults/LOC} \qquad \text{(Eq. 1-1)}$$

where Fault is an accidental condition that causes a functional unit to fail to perform its required function, and LOC is executable line of code. An RPFOM can be computed for an entire software system and/or its components (e.g., CSCIs). Four successive RPFOM equations, each representing a refinement to its predecessor, are given:

$$\text{RPFOM} = A \qquad \text{(Eq. 1-2)}$$

where "A" is a metric for System Application Type;

$$\text{RPFOM} = A * D \qquad \text{(Eq. 1-3)}$$

where "D" is a metric for System Development Environment;

$$\text{RPFOM} = A * D * S1 \qquad \text{(Eq. 1-4)}$$

where "S1" is a metric of Software Characteristics during software Requirements and Design Specification;

$$\text{RPFOM} = A * D * S1 * S2 \qquad \text{(Eq. 1-5)}$$

where "S2" is a metric of Software Characteristics during software Implementation. "A" is expressed as a baseline fault density, whereas "D," "S1," and "S2" are modification factors for which values can range from <1 (decreased fault density) to >1 (increased fault density).

The "S1" metric is derived from "SA (Anomaly Management), "ST" (Traceability), and "SQ" (Quality Review) metrics:

$$S1 = SA * ST * SQ \qquad \text{(Eq. 1-6)}$$

The "S2" metric is derived from "SL" (Language Type), "SM" (Modularity), "SX" (Complexity), and "SR" (Standards Review) metrics:

$$S2 = SL * SM * SX * SR \qquad \text{(Eq. 1-7)}$$

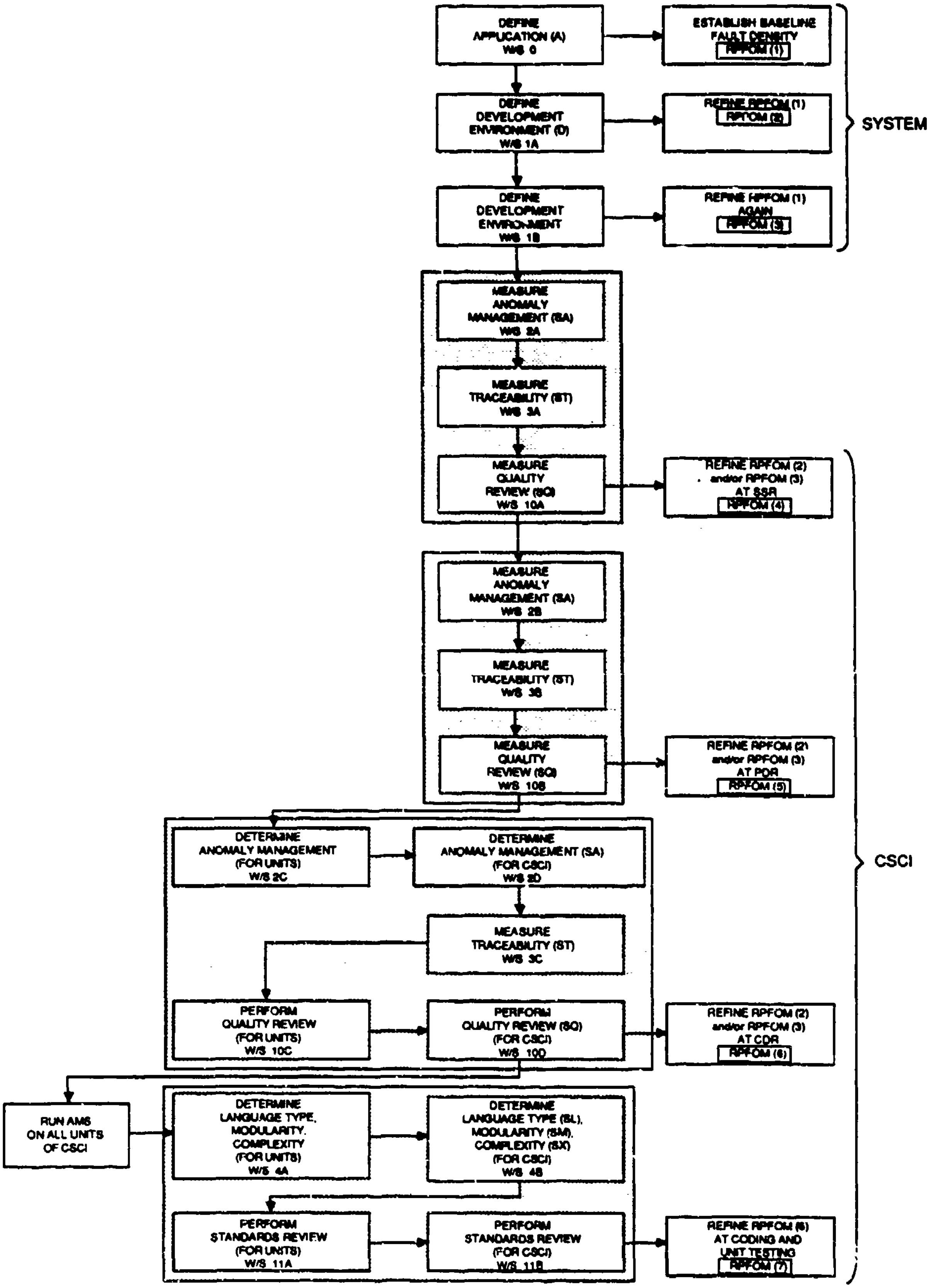

DEFINE APPLICATION (A) W/S 0
ESTABLISH BASELINE FAULT DENSITY
RPFOM (1)
DEFINE DEVELOPMENT ENVIRONMENT (D) W/S 1A
REFINE RPFOM (1)
RPFOM (2)
DEFINE DEVELOPMENT ENVIRONMENT W/S 1B
REFINE RPFOM (1) AGAIN
RPFOM (3)
SYSTEM
MEASURE ANOMALY MANAGEMENT (SA) W/S 2A
MEASURE TRACEABILITY (ST) W/S 3A
MEASURE QUALITY REVIEW (SQ) W/S 10A
REFINE RPFOM (2) and/or RPFOM (3) AT SSR
RPFOM (4)
MEASURE ANOMALY MANAGEMENT (SA) W/S 2B
MEASURE TRACEABILITY (ST) W/S 3B
MEASURE QUALITY REVIEW (SQ) W/S 10B
REFINE RPFOM (2) and/or RPFOM (3) AT PDR
RPFOM (5)
DETERMINE ANOMALY MANAGEMENT (FOR UNITS) W/S 2C
DETERMINE ANOMALY MANAGEMENT (SA) (FOR CSCI) W/S 2D
MEASURE TRACEABILITY (ST) W/S 3C
PERFORM QUALITY REVIEW (FOR UNITS) W/S 10C
PERFORM QUALITY REVIEW (SQ) (FOR CSCI) W/S 10D
REFINE RPFOM (2) and/or RPFOM (3) AT CDR
RPFOM (6)
CSCI
RUN AMS ON ALL UNITS OF CSCI
DETERMINE LANGUAGE TYPE, MODULARITY, COMPLEXITY (FOR UNITS) W/S 4A
DETERMINE LANGUAGE TYPE (SL), MODULARITY (SM), COMPLEXITY (SX) (FOR CSCI) W/S 4B
PERFORM STANDARDS REVIEW (FOR UNITS) W/S 11A
PERFORM STANDARDS REVIEW (FOR CSCI) W/S 11B
REFINE RPFOM (6) AT CODING AND UNIT TESTING
RPFOM (7)

FIGURE TS100-1   SUMMARY RPFOM DATA COLLECTION PROCESS FLOW

Computation of these metrics and metric components is presented in the following Sections. Table TS100-5 parallels this presentation and provides a summary of attributes for each RPFOM and the location in Tasks 101 through 104 for each worksheet and answer sheet.

100.5.1.1 Application RPFOM

The Application-type RPFOM (Eq. 1-2) is calculated for each test project. This baseline RPFOM, determined prior to initiation of software development, is an average fault density based on the principle application type of the test project (e.g., Airborne System). Metric Worksheet 0 provides a list of six application types for selection.

100.5.1.2 Development Environment RPFOM

The Development Environment RPFOM, which is a refinement of the baseline RPFOM, incorporates information pertaining to the software development environment of a system. This information, summarized in the "D" metric, should be available during a software pre-development phase of the life-cycle. Although this RPFOM is defined by a single expression (Eq. 1-3), one or two worksheets (either 1A, or 1A in combination with 1B) can be utilized to compute "D" depending on the level of detail of project environment data available.

Worksheet 1A provides a quick approximation of "D" based upon selection by the data collector of one of three development environment categories. Worksheet 1B provides a more precise determination of "D" based upon a checklist of 38 development environment characteristics:

$$\begin{aligned} D &= (0.109Dc - 0.04) / 0.014 && \text{if Embedded from W/S 1A} \\ &= (0.008Dc + 0.009) / 0.013 && \text{if Semi-detached from W/S 1A} \\ &= (0.018Dc - 0.003) / 0.008 && \text{if Organic from W/S 1A} \end{aligned}$$

where Dc = 1- (# characteristics in W/S 1B applicable to system) / 44).

100.5.1.3 Requirements and Design RPFOM

This RPFOM (Eq. 1-4), a refinement of the Development Environment prediction, incorporates information on software characteristics provided by system requirements and design documentation to determine the SA< ST, and SQ metric components of S11 (Eq. 1-6). Any of three sets of worksheets, each set specific to a particular Life-cycle phase, is used to derive these three metric components. Three Requirements and Design RPFOM values corresponding to SSR, PDR, and CDR were determined for each CSCI of the two software test projects in order to evaluate the usability of each set of work sheets. Derivation of SA, ST, SQ is summarized below.

Anomaly Management (SA):

The "SA" metric is equated to one of three values based on the value of "AM," which is derived from responses by data collectors to questions in Worksheets 2A (SSR), 2B (PDR), and 2C/2D (CDR) that apply to the capabilities of a system to respond to software errors and other anomalies:

AM = Number of "NO" responses/Total number of "YES" and "NO" responses

"SA" is then computed as follows:

$$\begin{aligned} SA &= 0.9 \text{ if } AM < 0.4 \\ &= 1.0 \text{ if } 0.6 \geq AM \geq 0.4 \\ &= 1.1 \text{ if } AM > 0.6 \end{aligned}$$

TABLE TS100-5 RPFOM ELEMENT INDEX

TASK	RPFOM	METRIC	PHASE	LEVEL	WORKSHEET	ANSWER SHEET
101	A	A	SRR	System	0	0
102	D	D	SRR	System	1A	0
				System	1B	0
103	S1	SA	SSR	CSCI	2A	A
			PDR	CSCI	2B	B
			CDR	Units	2C	C
				CSCI	2D	C
		ST	SSR	CSCI	3A	A
			PDR	CSCI	3B	B
			CDR	CSCI	3C	C
			SSR	CSCI	4A	A
			PDR	CSCI	4B	B
			CDR	Units	4C	C
				CSCI	4D	C
104	S2		C&UT	Units	5	
		SL	C&UT	Units	8D	D
				CSCI		
		SX	C&UT	Units		
				CSCI	9D	D
		SM	C&UT	Units	9D	D
				CSCI	10D	
		SR	C&UT	Units	11A	D
				CSCI	11B	

Traceability (ST)

A value for "ST" is selected by the data collector using Worksheets 3A (SSR), 3B (PDR), or 3C (CDR) which encompass traceability of requirements from system level through unit level.

Quality Review (SQ)

The "SQ" metric is equated to one of two values:

$$
\begin{aligned}
SQ &= 1.1 \text{ if DR / Total \# Y and N responses} > 0.5 \\
&= 1.0 \text{ if DR / Total \# Y and N responses} \leq 0.5
\end{aligned}
$$

DR is a count of "NO" responses from Worksheet 10A during Software Requirements Analysis (SSR), 10B during Preliminary Design (PDR), or 10C/10D during Detailed Design (CDR).

100.5.1.4 Implementation RPFOM

The Implementation RPFOM (Eq. 1-5), which represents a final refinement to the reliability prediction can be calculated at the CSCI, CSC, or unit level. Information on the software characteristics derived from source code during Coding and Unit Testing (C&UT) is incorporated to determine the SL, SM, SX, and SR components of S2 (Eq. 1-7). Unit-level metrics are collected for Worksheets 4A and 11A, and then summed for corresponding CSCs and CSCIs using Worksheets 4B and 11B. Data collection for this RPFOM should begin with the utilization of a tool such as the Automated Metrics System (AMS) for automatically collecting many of the unit-level metric elements. Values obtained from AMS hard-copy reports are transferred to answer sheets along with values for non-automated metrics as indicated in the worksheets. Derivation of SL, SM, SX, and SR components of S2 is summarized below.

Language Type (SL)

The "SL" metric is derived as follows:

$$SL = (HLOC/LOC) + (1.4\ ALOC/LOC)$$

where $HLOC$ = higher-order-language lines of code for CSCI
$ALOC$ = assembly language lines of code for CSCI
LOC = total executable lines of code for CSCI

Values for HLOC, ALOC and LOC are determined in Worksheets 4B based upon unit-level values for these variables in Worksheet 4A.

Complexity (SX)

The "SX" metric is derived as follows:

$$SX = (1.5a + b + 0.8c) / NM$$

where a = # units in CSCI with $sx \geq 20$
b = # units in CSCI with $7 \leq sx < 20$
c = # units in CSCI with $sx < 7$
NM = # units in CSCI
sx = complexity for unit

Values for a, b, and c are determined in Worksheet 9D.

Modularity (SM)

The "SM" Metric is derived as follows:

$$SM = (0.9u + w + 2x)/NM$$

where u = # units in CSCI with $MLOC \leq 100$
w = # units in CSCI with $100 < MLOC \leq 500$
x = # units in CSCI with $MLOC > 500$
$MLOC$ = executable lines of code in unit
NM = # units in CSCI = $u + w + x$

Values for u, w, and x are determined in Worksheet 9D.

Standards Review (SR)

The "SR" metric is derived as follows:

$$\begin{aligned} SR &= 1.5 \text{ if } \quad DF \geq 0.5 \\ &= 1.0 \text{ if } 0.5 > DF \geq 0.25 \\ &= 0.75 \text{ if } \quad DF < 0.25 \end{aligned}$$

where DF = (# "No" responses) / (# "No" + "Yes" responses)

from Worksheet 11D.

100.5.2 Data Collection Tasks

This Task Section of the Guidebook contains Tasks 101 through 104 for software reliability prediction. Each task refers to data collection procedures, worksheets and answer sheets appropriate to the lifecycle phases as follows:

a. Task 101: Project initiation
b. Task 102: Requirements
c. Task 103: Design
d. Task 104: Coding

100.5.3 Data Collection Procedures

Each task section identifies related data collection procedures. These procedures describe what data must be collected to use the software reliability prediction computations described in Section 100.5.1. Complementing these computations are the actual worksheets and answer sheets. The intended process then is for reliability engineers to use the worksheets in conjunction with these data collection procedures to collect data. That data will then be used when the engineer or analyst uses the prediction and estimation algorithms to determine a reliability number. A data collection procedure index for Tasks 101 through 104 is provided in Table TS100-6. The actual procedures and worksheets/answer sheets are in the appendices.

The utility of the metrics is based on their representation of the characteristics identified and the correlation or affect of these characteristics on software reliability. There is, however, another important aspect to the utility of the metrics. That is the economy of their use, i.e., the cost of collecting the data to calculate the metrics is an important consideration. Automated collection tools are essential for many of the measurements. Some measures, such as the ones which simply require classification, are easy to collect.

The task sections contain data collection procedures for all data required to calculate each metric. The procedures adhere to the following format:

Procedure Outline

1. Title: Identifies metric or data element this procedure relates to.

2. Prediction Parameters Supported: Identifies the higher level metric this procedure relates to.

3. Objectives: Objective of the title metric.

4. Overview: Provides overview of this metric.

5. Assumptions/Constraints: Describes any assumptions or constraints related to this metric.

6. Limitations: Describes any limitations to using the procedure or metric.

7. Applicability: Describes when the metric can be applied during the software life cycle.

8. Required Inputs: Identifies the required data for calculating the metric.

9. Required Tools: Identifies any required tools needed for data collection.

10. Data Collection Procedures: Provides step by step guidance on collecting the appropriate data.

11. Outputs: Describes output of procedure.

12. Interpretation of Results: Provides guidance on interpreting the results.

13. Reporting: Provides any required reporting format.

14. Form: Identifies any applicable forms for data collection or metric calculation.

15. Instructions: Specific instructions for using worksheets.

16. Potential/Plans for Automation: Describes potential and any known plans for automation of this data collection procedure.

17. Remarks: Allows any remarks/comments about metric.

The data required for these metrics is available during most DOD software developments. The Answer Sheets, in Appendix C, provide a phase orientation. These Answer Sheets can be copied and used to accumulate all of the answers for all worksheets by phase. Data collection is required. It involves applying worksheets to the typical documentation produced with MIL-STD 2167A, MIL-STD 490/483, and MIL-STD 1679 and automated tools to the code produced.

TABLE TS100-6. Task Data Collection Procedure Index

Task No.	Procedure Name	Procedure No.
101	Application Type (A)	0
102	Development Environment (D)	1
103	Anomaly Management (SA)	2
	Traceability (ST)	3
	Quality Review (SQ)	4
	Size Estimation (NL & SLOC)	5
	Fault Density	6
	Discrepancy Reports (DR)	7
104	Language Type (SL)	8
	Module Size (SM)	9
	Complexity (SX)	10
	Standards Review (SR)	11

100.5.3.1 Data Collection Procedures and Worksheets

Each task section references the procedures and worksheets used to collect metric data during development phases. Eleven different worksheets are applied to development products in different phases and at different levels of abstraction for reliability prediction. Instructions are contained in the procedures for filling out the worksheets. These worksheets are modeled after those documented in RADC TR 85-37.

One difference is that only those worksheet items pertinent to reliability prediction are included in this Guidebook. Any questions relating to definition, explanation, or application of these worksheets should be referred to RADC TR 85-37. Another difference is that the worksheets related to the Quality Review (SQ) metric and the Standard Review (SR) have been separated and reorganized in Tasks 104 and 103, respectively.

Terminology used in the worksheets generally is consistent with DoD-STD-2167A (e.g., CSCI, unit). The term "software" is used in a broad sense and refers both to the end product (code, data and documentation) and to the product in its current stage of evolutionary development. A glossary is in RADC TR 85-37.

100.5.3.2 Usage Standards

The instructions in Tasks 101 and 102 are applied once to each system. The instructions in Sections 103 and 104 are repeated for each CSCI of a system or for each unit of a CSCI.
Several standard options are available on the answer sheets. If a value cannot be determined for a numeric question, then the corresponding item on the answer sheet (and in the database) should simply be left blank. The remaining "NA" option for non-numeric questions is to be interpreted as "Not Applicable to Sample Project." This option should be utilized with caution, if at all. Its purpose is to flag questions which are not pertinent to the type of application (e.g., database) represented by the system under investigation.

An option of "UNK" (Unknown) has been provided for most of the non-numeric metric questions. This option should also be used with caution. It should be selected only if evidence from data sources (e.g., project documentation, interviews with developers, etc.) strongly suggests that a

"Yes" response to a question may be warranted, but cannot be chosen with certainty due to the unavailability of the information. In general, if a requirement, programming standard, etc., is not documented by the existing data sources, then the "No" option should be selected (a documented denial of a requirement, implementation, etc., is not absolutely necessary.)

TASK SECTION 101

SOFTWARE RELIABILITY PREDICTION BASED ON APPLICATION

TASK SECTION 101

SOFTWARE RELIABILITY PREDICTION BASED ON APPLICATION

101.1 Purpose

The purpose of Task 101 is to provide a method for predicting a baseline software reliability.

101.2 Documents Referenced

See Task Section 100.

101.3 General Procedures

Application Type (A)

Using Data Collection Procedure 0 and the corresponding Worksheet O, identify which application type the subject software represents, and assign the corresponding fault density to A which is derived from average fault density in Table TS101-1.

An initial RPFOM = A.

101.4 Detail to be Specified by the Procuring Authority

a. Statement of reliability requirements.

b. Define the software component level for prediction (different levels may be specified for each life cycle phase).

c. Define life cycle phases to be covered and prediction milestones.

d. Identify data collection procedures in this task.

e. Identify fault density/failure rate transformation procedure to be used.

101.5 Procedure

See Procedure No. 0.

TABLE TS101-1
BASELINE FAULT DENSITY AND FAILURE RATE VALUES

	FAULT DENSITY			FAILURE RATE	
APPLICATION	NO. OF SYSTEMS	TOTAL LINES TOTAL LINES OF CODE	AVG FAULT DENSITY BY SYSTEM	NO. OF SYSTEMS	OPTATIONAL FAILURE RATE
AIRBORNE	7	540,617	.013	1*	.08*
STRATEGIC	21	1,793,831	.009	5	0.0109
TACTICAL	5	88,252	.008	5	0.108
PROCESS CONTROL	2	140,090	.002	-	-
PRODUCTION CENTER	12	2,575,427	.009	4	0.198
DEVELOPMENTAL	6	193,435	.014	3*	11.8*
TOTAL/AVERAGE	53	5,331,652	.010	14	0.1

•These two values are failure rates during test, not included in total

TASK SECTION 102

SOFTWARE RELIABILITY PREDICTION BASED ON DEVELOPMENT ENVIRONMENT

TASK SECTION 102
SOFTWARE RELIABILITY PREDICTION BASED ON DEVELOPMENT ENVIRONMENT

102.1 Purpose

The purpose of Task 102 is to modify the baseline software reliability prediction calculated in Task 101 based on the type of development environment.

102.2 Documents Referenced

See Task Section 100.

102.3 General Procedures

Development Environment (D)

Using Data Collection Procedure 1 and the corresponding Worksheet 1A, identify which class of development environment is being used, and assign the corresponding value to the D metric.

Three classes of development environments have been provided:

a. Organic -- Software is being developed by a group that is responsible for the overall application (e.g., flight control software being developed by a manufacturer of flight control systems);

b. Semi-detached -- The software developer has specialized knowledge of the application area, but is not part of the sponsoring organization (e.g., network control software being developed by a communications organization that does not operate the target network); and

c. Embedded -- Software that frequently has very tight performance constraints being developed by a specialist software organization that is not directly connected with the application (e.g., surveillance radar software being developed by a group within the radar manufacturer, but not organizationally tied to the user of the surveillance information).

The baseline is the semi-detached environment. It is expected that the organic environment will generate software of lower fault density and the embedded environment software of greater fault density. The value, D_O, associated with each of these environments is given in Table TS102-1. The D metric is equated to the value selected:

$$D = D_O$$

Table TS102-1 Development Environment Metrics

ENVIRONMENT	METRIC Do (FAULT DENSITY MULTIPLIER)
Organic	.76
Semi-Detached	1.0
Embedded	1.3

If more specific data about the environment is available, calculate D using the Development Environment checklist of Worksheet 1B. This modified approach is based on specific organizational/personnel considerations, methods used, documentation to be produced, and tools to be used. Recalculate D as:

$$D = D_m$$

where D_m is calculated from the following:

$D_m = (.109\ D_c - .04)/.014$ if Embedded from Worksheet 1A

$D_m = (.008\ D_c + .009)/.013$ if Semi-detached from Worksheet 1A

$D_m = (.018\ D_c - .003)/.008$, if Organic from Worksheet 1A

where D_c = 1 minus the ratio of methods and tools applicable to system divided by 44, which is the total number of methods and tools in the checklist in worksheet 1B. D_c is a number between 0 and 1. D_m should never be less than .5 or greater than 2. If the calculations result in a number less than .5, set D_m = .5. If the calculations result in a value outside of these bounds, D_m should be adjusted accordingly:

$$\text{If } D_m < .5, \text{ set } D_m = .5$$
$$\text{If } D_m > 2, \text{ set } D_m = 2$$

PREDICTION

An updated prediction is calculated by:

$$RPFOM = A * D$$

102.4 Detail to be Specified by the Procuring Authority

a. Identify data collection procedures in this task.

b. Specify whether a generic or detail development environment factor is to be generated.

102.5 Procedure

See Procedure No. 1

TASK SECTION 103

SOFTWARE RELIABILITY PREDICTION BASED ON SYSTEM/SUBSYSTEM LEVEL SOFTWARE CHARACTERISTICS

TASK SECTION 103

SOFTWARE RELIABILITY PREDICTION BASED ON SYSTEM/SUBSYSTEM LEVEL SOFTWARE CHARACTERISTICS

103.1 Purpose

The purpose of Task 103 is to modify the baseline software reliability prediction calculated in Task 101 based on the software characteristics as they evolve during the requirements and design phases of a development.

103.2 Documents Referenced

See Task Section 100.

103.3 General Procedures

Software Characteristics (S)

The Software Characteristic metric, S, is a product (composite) of two submetrics;

$$S = S1*S2$$

Each one of which is in turn the product of several simple metrics as shown below:

Requirements and Design Representation Metric $S1 = SA*ST*SQ$

Anomaly Management (SA)	-	Optional
Traceability (ST)	-	Optional
Quality Review Results (SQ)	-	Optional

Software Implementation Metric $S2 = SL*SM*SX*SR$

Language (SL)	-	Recommended
Modularity (SM)	-	Optional
Complexity (SX)	-	Recommended
Standards Review (SR)	-	Recommended

S1 is described in this task. S2 is described in Task 104.

Note: These metrics and their corresponding impact on the reliability prediction are based on data collected from several projects. Those identified above as recommended have exhibited consistently good predictive results. Those listed as optional provide the reliability engineer additional information upon which to base the prediction but because either their predictive qualities have been inconsistent or they are based on a limited sample size, they are not recommended.

A description of the procedures for calculating these metrics follows.

Anomaly Management (SA) - Optional

a. Apply Data Collection Procedure 2 and the appropriate Worksheet 2A, 2B, 2C, 2D according to the phase of the project to the Requirements and Design Specifications of the subject project. Answer all questions related to Anomaly Management.

b. Calculate the Anomaly Management Metric using the following equation:

$$SA \begin{array}{l} = .9 \text{ if } AM < .4 \\ = 1 \text{ if } .6 \geq AM \geq .4 \\ = 1.1 \text{ if } AM > .6 \end{array}$$

where AM equals the score received using the worksheet

c. Note that SA is applicable at SSR, PDR, and CDR. The appropriate worksheet should be used depending on the reliability prediction milestone to calculate the AM metric.

Traceability (ST) - Optional

a. Apply Data Collection Procedure 3 and the corresponding Worksheets 3A, 3B, or 3C to the Requirements and Design Specifications and code of the subject project. Answer the traceability questions. If unable to answer, use following substeps and Worksheet 3D:

1. Itemize all specific requirements in Requirements Specification.

2. Count the number of individual requirements (NR). See Data Collection Procedure 5.

3. Review Design Specification and identify specific design statements that represent the fulfillment of a specific itemized requirement (a requirements derivative).

4. Count the number of requirements not addressed by design that should have been (DR). See Data Collection Procedures 6 and 7.

b. Calculate ST as follows:

$$ST = 1.1 \text{ if } \frac{NR - DR}{NR} < .9$$

$$= 1 \text{ if } \frac{NR - DR}{NR} > .9$$

Quality Review Results (SQ) - Optional

a. Apply Data Collection Procedure 4 and the corresponding Worksheets 4A, 4B, or 4C, 4D to the Requirements and Design Specifications of the subject project. Answer all questions related to Accuracy (AC.1), Completeness (CP.1), Consistency (CS.1, CS.2) and Autonomy (AU.1, AU.2).

b. Calculate the SR metric using the following equation:

$$SQ = 1.1 \text{ if } \underline{DR}/\text{Total \#Y and N responses} > .5$$

$$= 1 \text{ if } \underline{DR}/\text{Total \#Y and N responses} \leq .5$$

where DR is the number of "No" responses from Worksheet being applied.

PREDICTION

If these optional metrics are applied, then an updated prediction is calculated by:

$$RPFOM = A*D*S1$$

103.4 Detail to be Specified by the Procuring Authority

a. Identify data collection procedures in this task.

b. Specify that requirements shall be traced throughout development.

103.5 Procedures

Procedure No.	Title
2	Anomaly Management (SA)
3	Traceability (ST)
4	Quality Review (SQ)
5	Size Estimation (NR & SLOC)
6	Fault Density
7	Discrepancy Reports (DR)

TASK SECTION 104

SOFTWARE RELIABILITY PREDICTION BASED ON CSC/UNIT LEVEL CHARACTERISTICS

TASK SECTION 104

SOFTWARE RELIABILITY PREDICTION BASED ON CSC/UNIT LEVEL CHARACTERISTICS

104.1 Purpose

The purpose of Task 104 is to modify the baseline software reliability prediction calculated in Task 101 based on the Software characteristics as they evolve during the coding phase of a development.

(This task can only be specified if Task 103 is also specified. See Task 103 for algorithm for combining the individual factors computed in Task 104.)

104.2 Documents Referenced

See Task Section 100

104.3 General Procedures

For each of the following metrics calculate their influence on software reliability. Note that some metrics are recommended and some are optional. See Task 103 (Note) for an explanation of the optional metrics.

Language Type (SL) - Recommended

a. Identify the total number of lines of code (SLOC) in the system (estimated or actual), the number of assembly language lines (ALOC), and the number of higher order language lines (HLOC). Note precisely the units of measurement employed (e.g., lines of source code, executable lines of code, executable statements for HLOC), which may differ among automated tools available to extract these data. Use Data Collection Procedure 5 (See Task Section 103) and 8 and Data Collection Worksheet 8A.

b. Use the following equation to calculate the language metric.

$$SL = HLOC/SLOC + 1.4 * ALOC/SLOC$$

Modularity (SM) - Optional

a. Count the number of modules in the system (NM) and the lines of executable code in each module (SLOC(i)). Use Data Collection Procedure 9 and Metric Worksheet 9D.

b. Table TS104-1 illustrates the impact on the predicted reliability by the number of modules in each size category.

c. Use the following equation to calculate modularity:

$$SM = (.9u + w + 2x)/NM$$

where $NM = u + w + x$

Table TS104-1 Module Categories

SIZE CATEGORY	NUMBER OF MODULES IN SIZE CATEGORY	MODULARITY METRIC
$LOC \leq 100$	u	.9
$100 < LOC \leq 500$	w	1
$500 < LOC$	x	2

Complexity (SX) - Recommended

a. Apply a complexity measure to each module in the system (sx(i)). Use Data Collection Procedure 10 and Metric Worksheet 9D.

Note precisely the measure of complexity employed (e.g., total number of branches, McCabe's, etc.).

b. Use the following equation to derive system complexity multiplier:

$$SX = (1.5(a) + 1(b) + .8(c))/ NM$$

where

a = number of modules with a complexity (sx) ≥ 20.

b = number of modules with a complexity between 7 and 20:
$7 \leq sx < 20$

c = number of Modules with a complexity (sx) less than 7.

NM = the total number of modules = a + b + c.

Standards Review (SR) - Recommended

a. Apply Data Collection Procedure 11 and corresponding Woiksheet 11D to the code. Answer all questions related to SI.1, SI.2, SI.4, SI.5, MO.1, MO.2.

b. Calculate this metric as follows:

$$SR = 1.5 \text{ if } DF \geq 0.5$$
$$1 \text{ if } .50 > DF \geq .25$$
$$.75 \text{ if } DF < .25$$

PREDICTION

Based on the application of these metrics, an updated prediction is calculated by:

$$RP = A * D * S1 * S2$$

If optional metrics are not used then assign a value of 1 to their multiplier.

104.4 Detail to be Specified by the Procuring Authority

a. Identify Data Collection Procedures.

104.5 Procedures

Apply the following procedures for the collection of data necessary to support the calculation of the metrics.

Procedure No.	Title
6	Size Estimation (NR & SLOC)
8	Language Type (SL)
9	Module Size (SM)
10	Complexity (SX)
11	Standards Review (SR)

TASK SECTION 200

SOFTWARE TEST TECHNIQUE SELECTION

TASK SECTION 200
SOFTWARE TEST TECHNIQUE SELECTION

OVERVIEW OF METHODOLOGY

200.1 Purpose

The purpose of Task 200 is to describe general procedures for selecting and applying state-of-the-art software testing techniques.

200.2 Documents Referenced in Task 200

MIL-STD 2167A
RADC TR 83-11
RADC TR 84-53
MIL-STD 756B
MIL-STD 785B

200.3 General Procedures

The Test and Evaluation approach in any system development is a critical element of the Reliability Program and ultimately to the reliability of the fielded system.

There are two kinds of test and evaluation (T&E) in the system acquisition process: DT&E and OT&E. Their primary purposes are to identify, assess, and reduce the acquisition risks, to evaluate operational effectiveness and operational suitability, and to identify any deficiencies in the system. Adequate T&E must be performed before each major decision point to make sure that the major objectives of one phase of the system acquisition life cycle have been met before the next phase is begun. Quantitative data must be used to the maximum extent practical, to show that the major objectives have been met. Subjective judgement, relative to systems performance, must be minimized.

The following two sections describe DT&E and OT&E objectives and the relationship of DT&E with the test phases used in the Guidebook. Detailed information on T&E can be found in AFR 80-14.

Development Test and Evaluation

Through DT&E, it must be demonstrated that (1) the system engineering design and development are complete, (2) design risks have been minimized, and (3) the system will perform as required and specified. This involves an engineering analysis of the system's performance, including its limitations and safe operating parameters. The system design is tested and evaluated against engineering and performance criteria specified by the acquisition organization and the using command. DT&E addresses the logistic engineering aspects of the system design and may go on all through the life cycle. It may include testing not completed before the first major production decision. It may also involve testing product improvements or modifications designed to correct identified deficiencies or to reduce life cycle costs.

The types of testing that occur during DT&E are algorithm confirmation during code and unit testing through system testing. In other words, DT&E activities include very low level testing to high level testing. The testing objectives include verifying that algorithms will satisfy the requirements imposed on the software design, verifying that the design is a correct implementation of the specified requirements, verifying that the unit's logic and interfaces satisfies the design specifications, verifying that the computer software configuration item (CSCI) is a correct implementation of the specified design, verifying that all specified real-time and functional requirements are satisfied, and verifying that the system is in agreement with the system level specifications.

This guidebook provides detailed procedures in Task Section 200 to select and apply various state-of-the-art testing techniques during coding and unit testing, and Computer Software Component (CSC) integration and testing phases of DT&E.

Operational Test and Evaluation

OT&E is conducted, in conditions made as realistic as possible. throughout the system life cycle once a system has been developed. It is done to estimate (or to refine estimates of) a system's operational effectiveness and operational suitability in order to identify any operational deficiencies and the need for any modifications.

Through OT&E, the Air Force measures the system against the operational criteria outlined in pertinent program documentation (e.g., system operational concepts) developed by DoD. HQ USAF, and using and supporting commands. Information is provided on organizational structure, personnel requirements, doctrine, and tactics.

OT&E is used to provide data to verify operating instructions, computer documentation, training programs, publications, and handbooks. It uses personnel with the same skills and qualifications as those who will operate, maintain, and support the system when deployed.

Types of OT&E include initial OT&E (IOT&E) and follow-on OT&E (FOT&E). On certain programs, qualification OT&E (QOT&E) is conducted instead of IOT&E. For guidance on conducting OT&E, see AFM 55-43.

IOT&E is conducted before the first major production decision. It is done by the OT&E command or agency (hereafter called OT&E command) designated by HQ USAF. As a rule, it is done using a prototype, pre-production article or a pilot production item as the test vehicle.

FOT&E is that operational testing usually conducted after the first major production decision or after the first production article has been accepted. It may go on all through the remainder of the system life cycle. In this case, it may be done to refine estimates of operational effectiveness and suitability, to identify operational deficiencies, to evaluate system changes, or to reevaluate the system against changing operational needs.

The types of testing that occur during OT&E according to the test phases used in this handbook are system and mission testing. That is, OT&E activities include very high level testing. The testing objectives include verifying that the entire system meets its system level specifications and verifying that the entire system meets the requirements of the mission.

In each of these phases of testing it is important to utilize testing techniques and strategies that will result in a thorough evaluation of the system and maximum identification of errors.

Test Technique Selection

This Task Section, in conjunction with the Software Test Handbook, RADC-TR-84-53, provides the guidance for selecting effective test techniques.

It is not the intent to duplicate in this Guidebook the procedures and methodology described in RADC-TR-84-53. The procedures provided here highlight updates to the Software Test

Handbook based on data collection, analysis, and experimental evaluations. These procedures also provide the integration of test technique selection with Reliability Estimation.

200.3.1 Test Techniques

The testing techniques covered by the Software Test Handbook (STH) are listed in Table TS200-1. Definitions and descriptions are found in the STH.

200.3.2 Testing Technique Selection Considerations

Software testing techniques are selected using the tables in Task Section 201 through 203. The selection process is a matter of using the tables. A methodology based on the use of tables was adopted in RADC TR 84-53 because it simplified the selection process and condensed a large amount of information. This methodology has been revised and carried over into the present Guidebook. The tables can be updated easily as state-of-the-art software testing continues to evolve.

Each table entry represents many considerations. The original tables were constructed based on state-of-the-art software testing theory and a survey of current Air Force mission application test requirements. The following considerations were used to revise those tables according to actual results obtained from application of the six software testing techniques described in Appendix D.

200.3.2.1 Error Detection and Location Capability

The error detection and location capability of a test technique is the relative success at detection of specific error types. It is also the precision with which the technique locates the software error so that it can be understood, analyzed, and corrected efficiently.

The effectiveness of each of the static and dynamic testing techniques in the Guidebook is factored into the selection tables in Task Section 201 through 203. It is important to note that the first step in error location is a natural product of static technique testing, since these techniques find faults in the code; the remaining step is to trace the code fault back to its origination in the specification, design document, or coder (or even compiler) error. The dynamic techniques require additional effort. They detect failures, which represent errors. By using the failure to locate the fault in the code one can gain information needed to understand exactly what was coded incorrectly, (e.g., what the error was), and then trace it back to its phase of origin.

200.3.2.2 Cost and Schedule Impact

A testing technique may have several significant cost impacts. One cost is its development or acquisition cost and another is the cost of application. The application cost is a result of the computer resources required by the technique and the time and special skills or training required by the personnel applying the techniques. The ultimate cost benefit is the savings derived from a reliable software product during the operational phase. That is, both costly failures and the cost of correcting errors, once a system is fielded, will be reduced.

Actual test effort data, which represent the times testers took to reach stopping rules (completion criteria) for each technique has been collected during experimental application. It is recommended this data be collected within specific development environments. Test technique efficiency can be defined as the percent of errors found when the stopping rule is reached, divided by the time taken applying the technique until the stopping rule is reached.

Comparing application effort across techniques shows that the static techniques took much less time than the dynamic techniques in the test experiments. Of the three static techniques,

SOFTWARE TEST TECHNIQUES	
STATIC ANALYSIS	Code Review
	Error/Anamoly Detection
	Structure Analysis/Documentation
	Program Quality Analysis
	Input Spacing Partitioning
	A. Path Analysis
	B. Domain Testing
	C. Partition Analysis
	Data-Flow Guided Testing
DYNAMIC ANALYSIS	Instrumetation Based Testing
	A. Path/Structural Analysis
	B. Performance Measurement
	C. Assertion Checking
	D. Debug Aids
	Random Testing
	Functional Testing
	Mutation Testing
	Real-Time Testing
	SYMBOLIC TESTING
	FORMAL ANALYSIS

TABLE TS200-1 SOFTWARE TEST TECHNIQUES

error/anomaly detection and structure analysis were fully automated and thus took very little time; conversely, code review involved manually reviewing the code against a checklist. Branch testing utilized the most time on average, followed by random testing and functional testing. Similar to the single testing technique effort results, technique pairs of any two static techniques will also take less time than pairs of any two dynamic techniques.

Based on the experimental findings, at the unit level, the testing techniques rank in decreasing efficiency as follows:

a. Error and Anomaly Detection

b. Structure Analysis

c. Code Review

d. Functional Testing

e. Branch Testing

f. Random Testing

At the unit integration (CSC) level the ranking is similar with items d and e, above, interchanged. However, the only trend or relationship apparent here is that static techniques found a larger percentage of the known errors per unit time than did the dynamic techniques. As a general guideline, this points to recommending static analysis at the unit and CSC levels. Figure TS200-1 provides the relative effort to apply techniques.

200.3.2.3 Training

Some techniques are very difficult to understand and may require extensive training of personnel, using personnel with special skills (algebraic and symbolic analysis). Other techniques use skills that are related to the development process (e.g., design walk-throughs and code reviews).

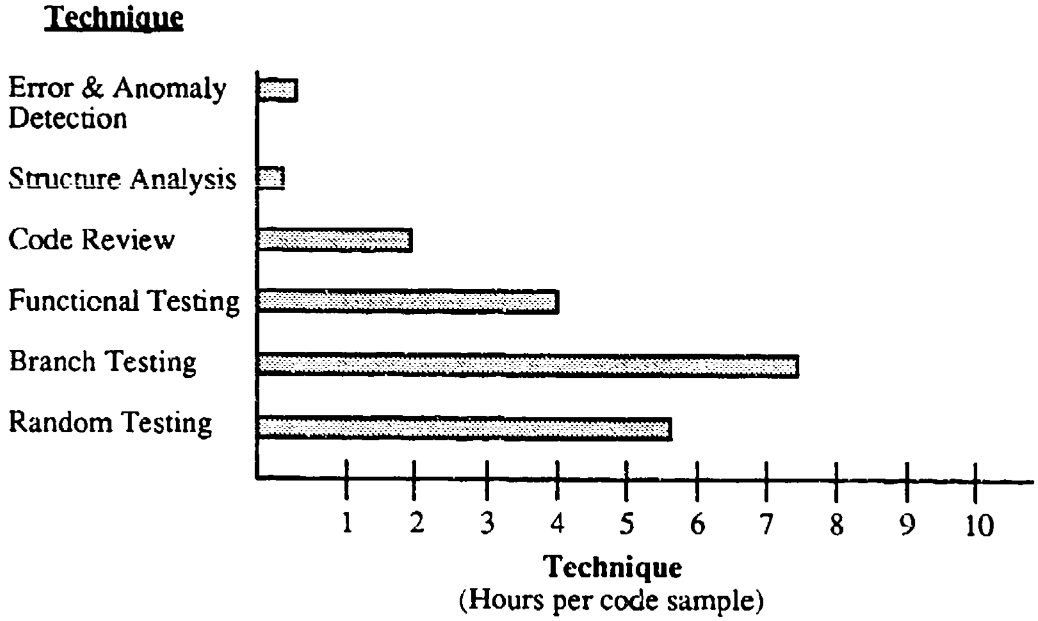
The effort studies showed it is not uniformly simple for testers to apply code review, as it has a subjective stopping rule. Also, when interviewed, the testers expressed that successful application of code review was dependent on their skill with and knowledge of the particular programming language(s) used in the code under test. These are not common skills to all testers, rather specific language skills.

Some techniques which are highly automated (e.g., structure analysis, the automated portion of error & anomaly detection) don't require tester training or expertise, but do require training and knowledge in the software tool which automates the technique. Contrarily, for code review and for each of the dynamic testing techniques, the technique effectiveness was shown to be highly dependent upon the tester applying the technique. Thus, acquiring good testers who have the requisite qualifications to apply the testing techniques is important.

200.3.2.4 Usage Constraints

The test and support tools are a major consideration to account for in the selection and application of testing techniques. Generic techniques are identified, but specific automated tools must be used. It is important to identify the availability of tools on the specific project hardware as part of the technique selection process. The tables in the guidebook rate generic software testing techniques, not specific software test tools.

FIGURE TS200-1 RELATIVE EFFORT TO APPLY TESTING TECHNIQUES



It is also important to note that structure analysis and error and anomaly detection are dependent upon the availability of source code analysis tools. Their application would be cost prohibitive if applied manually. An automated tool is also recommended to support the static phase of branch testing; otherwise, it could be verv time intensive to identify and select the various branch points for test case development. Finally, a random number generator is an essential support tool for the creation of test cases for the random testing technique.

Suitability of the testing techniques is also constrained by some characteristics of the software under test. This is discussed in 201.3 of Task Section 201.

200.3.2.5 Level of Human Interaction

The techniques included in the tables vary in the required level of human interaction. Human interaction must be considered at two levels: (a) a comparison of the amount of test engineer time required versus the potential benefits and (b) the degree of expertise need to effectively use the technique. The greater the relative amount of time or level of expertise, the lower the technique rating.

Regarding the first point above, test engineering time versus benefits (i.e., number of errors found) translates into efficiency. Section 200.3.2.2 shows the static testing techniques to be more efficient than the dynamic techniques to be more efficient than the dynamic techniques, and provides any empirically based order of decreasing efficiency for individual techniques.

The degree of expertise needed to effectively use the testing techniques is discussed in 200.3.2.3, above, and is summarized for individual techniques below. As a general rule for all techniques, the tester must be familiar with the methodology of the particular testing technique being applied, as per Appendix C.

a. **CODE REVIEW** - Requires skill with the source language; stopping rule is subjective and results are tester dependent.

b. **BRANCH TESTING** - Require some familiarity with the source language to associate input data with branch points; knowledge in a support tool; familiarity with the software specifications; familiarity with the input data.

c. **RANDOM TESTING**- Requires knowledge in a random number generator; familiarity with specifications.

d. **FUNCTIONAL TESTING** - Requires skill at interpreting/ translating textual requirements into test cases; familiarity with the software specifications; familiarity with the input data.

e. **ERROR AND ANOMALY DETECTION**- Requires knowledge in the testing tool.

f. **STRUCTURE ANALYSIS** - Requires knowledge in the testing tool.

The stopping rules in Task 201 seem to best reflect test application time excluding driver and on-line environment development. Additions for these activities can be separately estimated and added to the existing estimates, taking into account the driver complexity and number and types of tools used in setting up the test environment.

200.3.3 Alternate Selection Techniques

Procedures for selecting software testing techniques appropriate to software products and conditions, such as type of software or development phase are provided in Task Sections 201, 202, and 203. Extended discussions of the techniques are provided in the Software Test Handbook (STH).

The STH methodology provides three paths, shown in Figure TS200-2 that can be followed to determine appropriate software testing techniques. If possible, all three paths should be used to ensure that no relevant software techniques are omitted from consideration.

The three paths offered for the selection of testing techniques should not be confused with the three major applications of the guidebook described in section 1.3. All three paths are appropriate for each application.

In the first path (sec. 2.2 of the STH and Task Section 201) the selection of software testing techniques is based principally on the category of software being tested. For example, a real-time executive would require different testing approaches than those used for a post mission data analysis computer program.

The second path (sec. 2.3 of the STH and Task Section 202) uses information from the test phase and test objectives to select testing techniques. For example, an extensive real-time test would be appropriate after the real-time executive software had been successfully integrated and preliminary testing completed. The real-time test normally would not be appropriate during the module testing phase. The software acquisition lifecycle and associated test phases are described in section 5.0 of the STH.

The third path (sec. 2.4 of the STH and Task Section 203) selects testing techniques based on the knowledge of software error categories that are currently occurring or have previously occurred on similar software development projects.

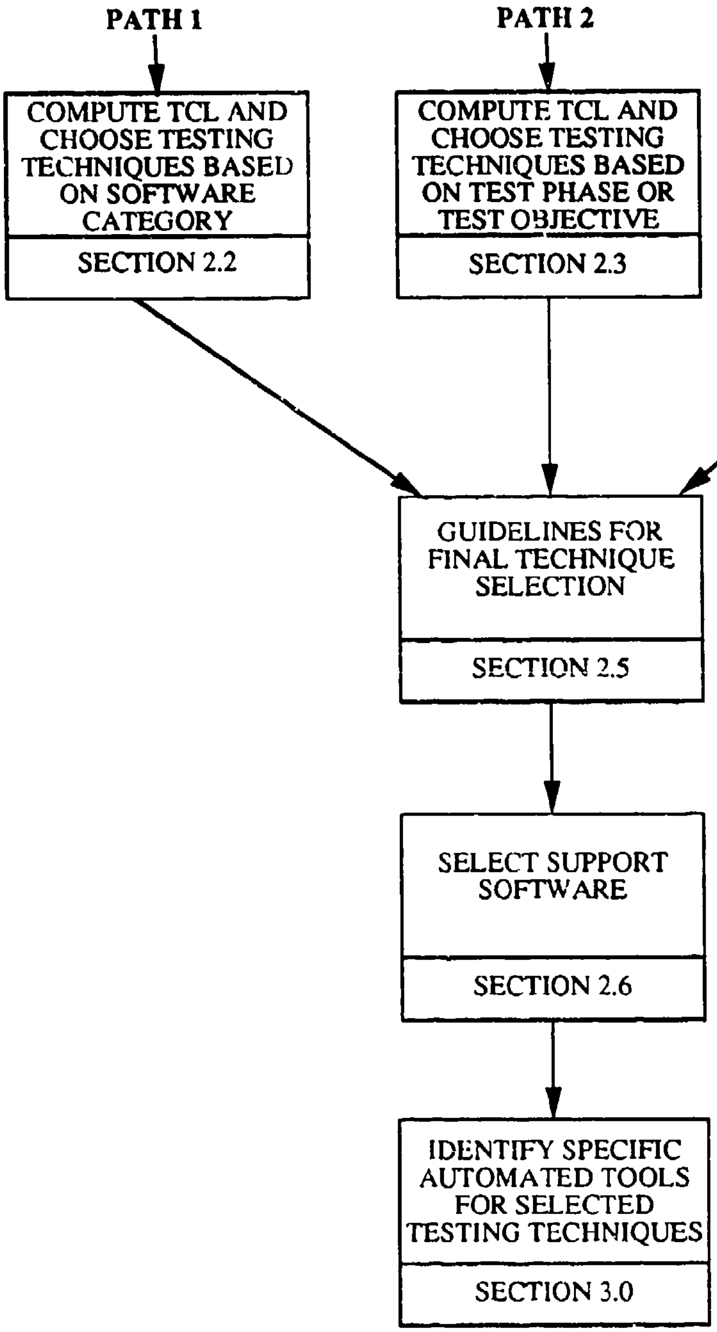

PATH 1
PATH 2
PATH 3
COMPUTE TCL AND CHOOSE TESTING TECHNIQUES BASED ON SOFTWARE CATEGORY
SECTION 2.2
COMPUTE TCL AND CHOOSE TESTING TECHNIQUES BASED ON TEST PHASE OR TEST OBJECTIVE
SECTION 2.3
CHOOSE TESTING TECHNIQUES BASED ON TYPES OF SOFTWARE ERRORS
SECTION 2.4
GUIDELINES FOR FINAL TECHNIQUE SELECTION
SECTION 2.5
SELECT SUPPORT SOFTWARE
SECTION 2.6
IDENTIFY SPECIFIC AUTOMATED TOOLS FOR SELECTED TESTING TECHNIQUES
SECTION 3.0

FIGURE TS200-2 THREE PATHS FOR SELECTING SOFTWARE TESTING TECHNIQUES

TASK SECTION 201

TEST SELECTION PATH 1

TASK SECTION 201

TEST SELECTION PATH 1

201.1 Purpose

The purpose of Task 201 is to describe procedures for selecting test techniques to be applied to a system based on the type of software (category) to be tested.

201.2 Documents Referenced

See Task Section 200

201.3 General Procedures

Follow general procedures outline in RADC TR 84-53, (STH), paragraph 2.2

Eighteen (18) software categories found in Table TS201-1, are used in RADC TR 84-53 as a basis for selecting testing techniques (Path 1). The categories contain groupings of similar complexity and criticality which were derived from a survey of Air Force testing practices and software engineering considerations. The results of those evaluations were used to recommend appropriate testing techniques.

Based on the category of software being developed and the required Testing Confidence Level (TCL), test techniques are recommended. The specific procedures are in the STH. The only changes based on recent empirical data is for simulation software. The technique ratings for simulation software are shown in Table 201-2. This data should be used in lieu of data presented in Fig. 2-8 in the STH.

201.4 Detail to be Specified by the Procuring Authority

a. Define Path 1 for selection process (Task 201)

TABLE TS201-1 SOFTWARE CATEGORIES

NO.	SOFTWARE CATEGORY	DESCRIPTION
1.	Batch (General)	Can be run as a normal batch job and makes no unusual hardware or input-output actions (e.g., payroll program and wind tunnel data analysis program). Small, throwaway programs for preliminary analysis also fit in this category.
2.	Event control	Does real-time processing of data resulting from external events. An example might be a computer program that processes telemetry data.
3.	Process control	Receives data from an external source and issues commands to that source to control its actions based on the received data.
4.	Procedure control	Controls other software; for example, an operating system that controls execution of time-shared and batch computer programs.
5.	Navigation	Does computation and modeling to computer information required to guide an airplane from point of origin to destination.
6.	Flight dynamics	Uses the functions computed by navigation software and augmented by control theory to control the entire flight of an aircraft.
7.	Orbital dynamics	Resembles navigation and flight dynamics software, but has the additional complexity required by orbital navigation, such as a more complex reference system and the inclusion of gravitational effects of other heavenly bodies.
8.	Message processing	Handles input and output messages, processing the text or information contained therein.
9.	Diagnostic software	Used to detect and isolate hardware errors in the computer in which it resides or in other hardware that can communicate with the computer.
10.	Sensor and signal processing	Similar to that of message processing, except that it required greater processing, analyzing, and transforming the input into a usable data processing format.
11.	Simulation	Used to simulate and environment, mission situation, other hardware, and inputs from these to enable a more realistic evaluation of a compute program or a piece of hardware.
12.	Database management	Manages the storage and access of (typically large) groups of data. Such software can also often prepare reports in user-defined formats, based on the contents of the database.
13.	Data acquisition	Receives information in real-time and stores it in some form suitable for later processing; for example, software that receives data from a space probe and files
14.	Data presentation	Formats and transforms data, as necessary, for convenient and understandable displays for humans. Typically, such displays would be for some screen presentation.

TABLE TS201-1 SOFTWARE CATEGORIES

NO.	SOFTWARE CATEGORY	DESCRIPTION
15.	Decision and planning aids	Uses artificial intelligence techniques to provide an expert system to evaluate data and provide additional information and consideration for decision and policy makers.
16.	Pattern and image processing	Used for computer image generation and processing. Such software may analyze terrain data and generate images based on stored data.
17.	Computer system software	Provides services to operational computer programs (i.e., problem oriented).
18.	Software development tools	provides services to aid in the development of software (e.g., compilers, assemblers, static and dynamic analyzers).

Notes:
The numbers in this table are related to the test confidenc^ level (TCL) computed using table 2-3. Select testing techniques that are rated ≤ TCL.

Path/Structural Analysis rating based on branch testing.

	STATIC ANALYSIS									DYNAMIC ANALYSIS										
TESTING TECHNIQUES	Code Reviews	Error/Anomaly Detection	Structure Analysis/Documentation	Program Quality Analysis	Input Space Partioning	A. Path Analysis	B. Domain Tesing	C. Partition Analysis	Data-Flow Guided Testing	Instrumentation-Based Testing	A. Path/Structural Analysis	B. Performance Measur^ments	C. Assertion Checking	D. Debug Aids	Random Testing	Functional Tesing	Mutation Testing	Real-Time Testing	SYMBOLIC TESTING	FORMAL ANALYSIS
SOFTWARE CATEGORY																				
BATCH (GENERAL)	0	1	2	2		2	1	2	3		2	1	2	2	3	0	3		3	3
EVENT CONTROL	0	0	1	1		1	1	3	3		2	1	2	1	2	0	3	2	3	3
PROCESS CONTROL	0	0	0	1		1	0	3	2		1	1	2	1	2	0	3	1	3	3
PROCEDURE CONTROL	0	0	0	0		1	0	3	2		1	1	1	0	2	0	2	1	2	3
NAVIGATION	0	0	0	2		2	2	3	2		2	2	2	2	3	0	3	1	3	3
FLIGHT DYNAMICS	0	0	0	1		2	1	2	2		2	2	2	2	2	0	3	0	3	3
ORBITAL DYNAMICS	0	0	0	1		2	1	1	2		2	2	1	0	2	0	2	1	2	3
MESSAGE PROCESSING	0	0	1	2		2	1	2	3		2	2	2	1	2	0	3	2	3	3
DIAGNOSTIC SOFTWARE	0	0	0	1		0	0	2	3		0	2	2	1	3	0	3	3	3	3
SENSOR & SIGNAL PROCESSING	0	0	0	0		0	0	2	2		1	1	2	0	2	0	2	1	3	3
SIMULATION	1	1	3	2		0	0	2	3		2	1	2	0	2	0	2	1	3	3
DATABASE MANAGEMENT	0	0	1	1		2	0	1	2		1	1	2	1	2	0	3		3	3
DATA ACQUISITION	0	0	0	2		2	0	3	3		2	1	2	1	2	0	3	1	3	3
DATA PRESENTATION	0	0	0	1		1	0	2	3		2	1	2	0	2	0	3	1	2	3
DECISION & PLANNING AIDS	0	0	0	1		1	1	1	2		1	2	2	1	2	0	3		2	3
PATTERN & IMAGE PROCESSING	0	0	0	1		1	0	1	2		1	2	2	0	2	0	3	1	2	3
COMPUTER SYSTEM SOFTWARE	0	0	0	1		1	1	1	2		1	2	2	0	2	0	2	1	2	3
SOFTWARE DEVELOPMENT TOOLS	0	0	0	1		1	1	1	1		1	2	2	0	2	0	3		3	

NOTE: Blank entry indicates the testing technique is not applicable to the software category

TABLE TS201-2 SOFTWARE CATEGORIES AND TESTING TECHNIQUES

TASK SECTION 202

TEST SELECTION PATH 2

TASK SECTION

TEST SELECTION PATH 2

202.1 Purpose

The purpose of Task 202 is to describe procedures for selecting test techniques to be applied to a system based on Test Phase and Test Objective.

202.2 Documents Referenced

See Task Section 200

202.3 General Procedures

Follow general procedures outlined in RADC TR-84-53 (STH), paragraph 2.3.

202.4 Detail to be Specified by Procuring Authority

a. Define Path 2 for Selection process (Task 202)

TASK SECTION 203

TEST SELECTION PATH 3

TASK SECTION 203

TEST SELECTION PATH 3

203.1 Purpose

The purpose of Task 203 is to describe procedures for selecting test techniques to be applied to a system based on Types of Errors to be found.

203.2 Documents Referenced

See Task Section 200.

203.3 General Procedures

Follow general procedures outlined in RADC-TR-84-53 (STH), paragraph 2.4. These procedures have been updated here for six common test techniques.

203.3.1 Test Effectiveness and Coverage

The procedures in RADC-TR-84-53 have been updated based on data collection. These procedures are based on test effectiveness (number of errors found) and test coverage (number of branches executed) which provide excellent criteria for the selection of testing techniques. Both of these criteria can be factored by test effort to assess test efficiency. Tables are provided in this section which include rankings of six testing techniques based on these factors, both singly and in pairs, for unit level testing and separately for CSC level testing. Further information can be found in RADC TR 84-53. The following paragraphs summarize the contents of these tables and describe how to use them to select candidate testing techniques based on these factors. Separate discussions of test effort and test coverage are provided in Data Collection Procedure No. 15 and 17 (Task Section 300), respectively.

From experimental results, it is apparent that unit testing found more errors on average than CSCI and system level testing of the same software. Therefore, the recommendation to require and formalize unit testing is made.

Code Review was the most effective test technique at the unit level, and Branch testing was the most effective test technique at the CSC level. The statistical analyses then showed that the differences seen were influenced by differences among the testers. Thus, a primary concern is to utilize good testers. Time constraints and partial CSC testing also factor into the data. The data tend to support the effort estimates in the stopping rules, making them useful as a guideline.

Use Tables TS203-1 and TS203-2 to select testing techniques at the unit level based on test effectiveness and test coverage. These tables list each of the six testing techniques in decreasing order of Error Detection Efficiency. They are also grouped into four categories with Category 1 techniques being the most efficient.

Detection Efficiency is the ratio of Average Effort over % Errors Found. All other columns in the tables identify the relative ranking of the testing techniques for that particular attribute, including Average Coverage and Coverage Efficiency. Coverage Efficiency is the ratio of Average Effort over Average Coverage.

Use Table TS203-1 to select single testing techniques. Use Table TS203-2 to select pairs of testing techniques at the unit level. This latter table identifies which techniques are strongest when

TABLE TS203-1 SINGLE TEST TECHNIQUE RANKINGS BY EFFORT, EFFECTIVENESS, COVERAGE AND EFFICIENCY (UNIT LEVEL)

TEST TECHNIQUE	STOPPING RULE (HOURS)	AVERAGE EFFORT	% OF ERRORS FOUND	DETECTION EFFICIENCY RANKING	% AVERAGE COVERAGE	COVERAGE EFFICIENCY RANKING
ERROR/ANOMALY DETECTION	6	0.97 (2)	.29 (2)	(1)	—	—
STRUCTURE ANALYSIS	4	0.66 (1)	.03 (6)	(2)	—	—
CODE REVIEW	8	13.32 (4)	.36 (1)	(3)	—	—
FUNCTIONAL TESTING	16	12.34 (3)	.23 (4)	(4)	.81 (2)	(1)
BRANCH TESTING	29	17.03 (6)	.25 (3)	(5)	.93 (1)	(3)
RANDOM TESTING	22	13.97 (5)	.19 (5)	(6)	.81 (2)	(2)

TABLE TS203-2 PAIRED TEST TECHNIQUE RANKINGS BY EFFORT, EFFECTIVENESS, COVERAGE AND EFFICIENCY (UNIT LEVEL)

TESTING TECHNIQUES											
CATEGORY	ERROR/ANOMALY DETECTION	CODE REVIEW	FUNCTIONAL TESTING	BRANCH TESTING	RANDOM TESTING	STRUCTURE ANALYSIS	AVERAGE EFFORT	% OF ERRORS FOUND	DETECTION EFFICIENCY RANKING	% AVERAGE COVERAGE	COVERAGE EFFICIENCY RANKING
1	•					•	2.17(1)	.39(9)	(1)	—	—
	•	•					4.95(3)	.64(1)	(2)	—	—
		•				•	4 67(2)	.44(7)	(3)	—	—
2	•		•				24.27(10)	.62(2)	(4)	.81(7)	(1)
		•			•		24 19(9)	.58(4)	(5)	.81(7)	(1)
	•				•		23.03(5)	.53(5)	(6)	.81(7)	(1)
	•			•			25.89(12)	.59(3)	(7)	.93(3)	(1)
		•	•				23.37(6)	.52(6)	(8)	.81(7)	(1)
3			•		•		23.70(7)	39(10)	(9)	.81+(6)	(1)
			•			•	23.99(8)	37(12)	(10)	.81(7)	(1)
				•	•		26.05(13)	38(11)	(11)	.93+(1)	(1)
				•		•	25.61(11)	33(14)	(12)	.93(3)	(1)
					•	•	22.75(4)	27(15)	(13)	.81(7)	(1)
4		•		•			51.44(15)	43(8)	(14)	93(3)	(11)
			•	•			50 46(14)	36(13)	(15)	93+(1)	(11)

+ Indicates probable higher coverage due to combined dynamic techniques.

used together. Locate single techniques or technique pairs in the column(s) of the table which are of interest, and select one or more techniques in each column which have the lowest (best) ranking. Enter these selections on Worksheet 7A in Appendix B.

For example, for single techniques you might fill out Worksheet 7A might be filled out as follows (reference example in Table TS203-3):

a. Structure Analyses is selected because it requires the least Average Effort, and can be applied with Error/Anomaly Detection using the same tool.

b. Error/Anomaly Detection is selected because it is the most efficient at detecting errors.

c. Branch Testing is selected because it produces the greatest coverage.

This is an example, depending on the specifics of the situation other choices might be made. Selections using considerations that are applicable to the software testing requirements of the specific project should be made.

Table TS203-3 also provides an example of using Table TS203-2 to select paired testing techniques at the CSC level. The lowest (best) rankings in Table TS203-2 are established for pairs of techniques which complement each other well according to the column heading.

Rankings in the right most two columns of Table TS203-2 require discussion. Static techniques are not ranked for test coverage because they do not cause the software to be executed. For % Average Coverage, all of the technique pairs are similarly ranked among Categories 2, 3 and 4. For Coverage Efficiency all technique pairs in Categories 2 and 3 are virtually identical and are ranked "1". Category 4 pairs are virtually identical, also, but worse than the rest and are ranked "11". This implies that you would not select pairs of techniques based on % Average Coverage and that Coverage Efficiency pairs in Category 4 would be ignored for this purpose.

For example, for paired techniques you might fill out Worksheet 7A as follows (reference example in Figure TS203-3):

a. The first three pairs are selected for their high Detection Efficiency rating.

b. The second and fourth pairs are selected for the high % Errors Found.

Note that these selections effectively choose Error/Anomaly Detection, Code Review, Functional Testing and Structure Analysis.

In comparison to the earlier single technique selections, Structure Analysis and Error/Anomaly Detection are reconfirmed and Code Review and Functional Testing are added. You can adopt either or both of these selections as candidates for the unit test level.

Continue on to Figures TS203-4 and TS203-5 with Worksheet 7B (Appendix B) in order to complete your testing technique selections based on test effectiveness and test coverage for CSC level testing. This is accomplished in the same manner as for the unit test level. When you have completed these worksheets then enter your selections in Worksheet 7C (Appendix B).

Continue those selections with to selected testing techniques based on error category for both the unit and CSC test levels, following path 3, paragraph 2.4, in the STH.

TABLE TS203-3

WORKSHEET 7A EXAMPLE

TEST TECHNIQUE SELECTION BASED ON TEST EFFECTIVENESS/TEST COVERAGE (UNIT LEVEL)

TEST TECHNIQUE	SELECTED (X)	
	Single	Paired
Error/Anomaly Detection		
Code Review		
Branch Testing		
Functional Testing		
Structure Analysis		
Random Testing		

TABLE TS203-4 SINGLE TEST TECHNIQUE RANKINGS BY EFFORT, EFFECTIVENESS, COVERAGE AND EFFICIENCY (CSC LEVEL)

TEST TECHNIQUE	STOPPING RULE (HOURS)	AVERAGE EFFORT*	% OF ERRORS FOUND	DETECTION EFFICIENCY RANKING	% AVERAGE COVERAGE	COVERAGE EFFICIENCY RANKING
ERROR/ANOMALY DETECTION	12	2.75 (1)	.25 (4)	(1)	—	—
CODE REVIEW	16	6.25 (3)	.35 (2)	(2)	—	—
BRANCH TESTING	58	13.06 (5)	.68 (1)	(3)	.76 (1)	(2)
FUNCTIONAL TESTING	32	8.81 (4)	.33 (3)	(4)	.68 (2)	(1)
STRUCTURE ANALYSIS	18	2.75 (1)	.10 (6)	(5)	—	—
RANDOM TESTING	44	13.06 (5)	.17 (5)	(6)	.62 (3)	(3)

* DRIVER DEVELOPMENT = 21.13

TABLE TS203-5 PAIRED TEST TECHNIQUE RANKINGS BY EFFORT, EFFECTIVENESS, COVERAGE AND EFFICIENCY (CSC LEVEL)

TESTING TECHNIQUES												
CATEGORY	ERROR/ANOMALY DETECTION	CODE REVIEW	FUNCTIONAL TESTING	BRANCH TESTING	RANDOM TESTING	STRUCTURE ANALYSIS	AVERAGE EFFORT	% OF ERRORS FOUND	DETECTION EFFICIENCY RANKING	% AVERAGE COVERAGE	COVERAGE EFFICIENCY RANKING
1	●	●					8.50(2)	.53(7)	(1)	—	—
		●				●	8.50(2)	.53(7)	(1)	—	—
	●					●	5.50(1)	.32(1)	(3)	—	—
2		●	●				15.56(4)	.48(6)	(4)	.68(7)	(1)
				●		●	31.13(10)	.81(14)	(5)	.76(3)	(3)
	●			●			31.13(10)	.81(14)	(5)	.76(3)	(3)
		●			●		17.00(5)	.43(3)	(7)	.62(10)	(2)
	●		●				28.13(5)	.68(10)	(8)	.66(7)	(5)
			●			●	28.13(6)	.61(9)	(9)	.68(7)	(5)
3		●		●		●	44.44(12)	.74(12)	(10)	.76(3)	(9)
			●	●			47 00(13)	74(12)	(11)	.76+(1)	(10)
	●				●		31 00(8)	.43(3)	(12)	62(10)	(7)
					●	●	31.00(8)	.43(3)	(12)	62(10)	(7)
				●	●		51.25(15)	69(11)	(14)	76+(1)	(11)
4			●		●		47 00(13)	36(2)	(15)	68+(6)	(12)

+ Indicates probable higher coverage due to combined dynamic techniques

203.3.2 Error Category

The procedures and tables in the STH are pudated in the following paragraphs. Use the tables in this section to select testing techniques if you know the categories of software errors that are occurring in the software to be tested, or that have occurred in similar projects and are likely to occur in this one. The categories of software errors used in this table are based on the results of testing experiments and studies. It is cautioned that this classification is descriptive and limited due to the small number of error observations.

Locate the software error categories in Table TS203-6 that are either occurring or are predicted to occur. Note the software testing techniques that are rated as effective against these errors. Also note their relative effectiveness. The ratings are H (high), M (moderate), L (low) and are a measure of the effectiveness of a technique at detecting software errors in specific categories. Add these techniques to the candidate list. In most cases, choose only the most highly rated techniques in that row. If there are techniques rated H, use only those techniques; if the highest ratings are M, use those. However, if the highest rating is L, it is doubtful that the technique will have a significant effect against this error type. The effectiveness ratings are used to aid in the selection process. If an error category has an H in its row, then that technique is very effective at detecting that type of error. The H, M, L ratings are based only on this criterion. Other factors such as cost and ease of use are not included and must be considered separately.

Record the candidate techniques by indicating their ratings on Worksheet 7A (Appendix B) in the third column for error category. With the first two columns for test effectiveness and test coverage completed also, you can complete the list of recommended testing techniques.

Table TS203-7 includes an example of how to use Table TS203-6. In this example, it is desired to select the highest rated testing techniques for detecting the following types of errors:

a. Logic errors
b. Data validation errors
c. Data handling errors
d. Interface errors

These criteria, then, select the following techniques:

a. Code review.
b. Functional testing.
c. Branch testing.

When the selections based on error category have been completed, then indicate the selected testing techniques in Worksheet 7C as shown in the example in Figure TS203-8. Copies of these worksheets are provided for your use in Appendix B.

Table TS203-6 ERROR CATEGORIES

EFFECTIVENESS OF TESTING TECHNIQUE:

H = HIGH
M = MODERATE
L = LOW

	STATIC			DYNAMIC		
TESTING TECHNIQUES	CODE REVIEW	ERROR/ANOMALY DETECTION	STRUCTURE ANALYSIS	RANDOM TESTING	FUNCTIONAL TESTING	BRANCH TESTING
COMPUTATIONAL ERRORS	L	L		L	L	L
LOGIC ERRORS	H	M	L		L	M
DATA INPUT ERRORS				L		
DATA VERIFICATION ERRORS	L				L	
DATA HANDLING ERRORS	H			M	H	H
DATA OUTPUT ERRORS				L		L
INTERFACE ERRORS	M			M	H	H
DATA DEFINITION ERRORS	L	H				
DATA BASE ERRORS						
OTHER ERRORS	L			L	L	L

TABLE TS203-7
EXAMPLE SELECTION BY ERROR CATEGORIES

EFFECTIVENESS OF TESTING TECHNIQUE

H = HIGH
M = MODERATE
L = LOW

	STATIC			DYNAMIC		
TESTING TECHNIQUES	CODE REVIEW	ERROR/ANOMALY DETECTION	STRUCTURE ANALYSIS	RANDOM TESTING	FUNCTIONAL TESTING	BRANCH TESTING
COMPUTATIONAL ERRORS	L	L		L	L	L
LOGIC ERRORS	[✓ H]	M	L		L	M
DATA INPUT ERRORS				L		
DATA VERIFICATION ERRORS	L				L	
DATA HANDLING ERRORS	[✓ H]			M	[✓ H]	[✓ H]
DATA OUTPUT ERRORS				L		L
INTERFACE ERRORS	M			M	[✓ H]	[✓ H]
DATA DEFINITION ERRORS	L	H				
DATA BASE ERRORS						
OTHER ERRORS	L			L	L	L

Circle each occurrence of the highest rating available opposite the error categories of interest.

TABLE TS203-8
EXAMPLE WORKSHEET 7C FOR
ERROR CATEGORY (UNIT LEVEL

Worksheet 7C
Test Technique Selection For
Test Effectiveness, Test Coverage and Error Category
(Unit Level)

SOFTWARE TEST TECHNIQUES		TEST EFFECTIVENESS		TEST COVERAGE		ERROR CATEGORY	NOTES/ COMMENTS
		7A	7B	7A	7B	7 E	
STATIC	CODE REVIEWS					X	
	ERROR/ANOMALY DETECTION						
	STRUCTURE ANALYSIS/DOC.						
DYNAMIC	RANDOM TESTING						
	FUNCTIONAL TESTING					X	
	BRANCH TESTING					X	

Enter an 'X' opposite the testing technique(s) selected from Worksheets 7A and 7B.

203.3.3 Combined Testing Technique Selections

After selecting testing techniques based on test effectiveness, test coverage and error category in, you will have completed Worksheet 7C. Now, you should evaluate the combined effect of these different selection criteria before making your final selections. Tables TS203-9 and TS203-10 illustrate this step using the examples which have been given.

Note that in our particular example, all testing techniques except random testing are recommended at both the unit and CSC level. At the unit level Figure TS203-9 shows that each of these five techniques are recommended twice, so there is no general preference of any one of them - preference is only on the basis of their original selection. That is, all but branch testing are recommended for greatest test effectiveness (no. errors found), where as branch testing is recommended for highest test coverage (no. branches executed). Code review, functional testing and branch testing are recommended for the specific error categories used for their selection. The optimum choice is to select all of the recommended techniques.

At the CSC level, a different pattern emerges in our particular example. Figure TS203-10 shows that code reviews are recommended four times, then functional testing three times and error/anomaly detection twice -- the others just once. Here, if your test budget were small or the mission of the software not critical, then you might select only the three most recommended techniques. Again, as at the unit level, the optimum choice is to select all of the recommended techniques.

203.3.4 Guidelines for Final Selection of Testing Techniques

If all three criteria for testing technique selection are evaluated (i.e., test effectiveness, test coverage and error category) worksheet 7C will be similar to those shown. Before arriving at a final selection of techniques from the candidate list, the following points should be considered. These guidelines are general and each situation should be considered unique, in that no general set of guidelines can be effective in all cases. The judgments and evaluations are qualitative; it is not possible to provide firm guidelines and precise methods of evaluation. Discussions with experienced software test engineers who have used the selected techniques will prove valuable.

203.3.4.1 Technique Suitability

Answers to the following questions will help you to determine suitability of the testing techniques you have selected to the particular software development environment. Is the technique applicable to this specific environment? Are there any special considerations that make this testing especially suitable or completely invalid? What are the strengths of the technique in this environment and are they appropriate here?

The following feedback from testers provides useful observations on the suitability of the various techniques in different settings.

a. **Random testing**: not suitable if input data are mostly binary or string data (unless there is a realistic operational profile for string data). Since random testing often involves more test cases than other techniques, testers found the error detection tedious.

b. **Code review**: suitable only if the tester is familiar with the implementation language, and from a practical standpoint only on manageable-sized sections of code. One of four testers disliked the formality of the checklist, and found it difficult to judge when the "intermediate stopping rule" of a given item on the checklist had been satisfied.

TABLE TS203-9
EXAMPLE COMPLETED

Worksheet 7C
Test Technique Selection For
Test Effectiveness, Test Coverage and Error Category
Unit Level

SOFTWARE TEST TECHNIQUES		TEST EFFECTIVENESS		TEST COVERAGE		ERROR CATEGORY	NOTES/ COMMENTS
		7A	7B	7A	7B	7 E	
STATIC	CODE REVIEWS		X			X	
	ERROR/ANOMALY DETECTION	X	X				
	STRUCTURE ANALYSIS/DOC.	X	X				
DYNAMIC	RANDOM TESTING						
	FUNCTIONAL TESTING		X			X	
	BRANCH TESTING			X		X	

Enter an 'X' opposite the testing technique(s) selected from Worksheets 7A and 7B.

TABLE TS203-10
EXAMPLE COMPLETED

Worksheet 7G
Test Technique Selection For
Test Effectiveness, Test Coverage and Error Category
(CSC Level)

SOFTWARE TEST TECHNIQUES		TEST EFFECTIVENESS		TEST COVERAGE		ERROR CATEGORY 7E	NOTES/ COMMENTS
		7C	7D	7C	7D		
STATIC	CODE REVIEWS	X	X		X	X	Rank 1
	ERROR/ANOMALY DETECTION	X	X				Rank 3
	STRUCTURE ANALYSIS/DOC.		X				
DYNAMIC	RANDOM TESTING						
	FUNCTIONAL TESTING			X	X	X	Rank 2
	BRANCH TESTING					X	

Enter an 'X' opposite the testing technique(s) selected from Worksheets 7C, 7D, and 7E.

c. **Branch testing**: not suitable for code with few branches. However, there is strong tester acceptance of this technique because it provides a concrete measure of one aspect of their testing progress.

d. **Functional testing**: suitability is dependent upon quality of the functional specifications. In particular, the stopping rule for functional testing is based on testing all functions in the specification; if the specification is vague then applying functional testing and knowing when you've reached the stopping rule becomes especially difficult. Two testers believed the results of functional testing were quite dependent upon tester expertise, and noted boundary checking for error data as an example that tester skill may impact.

e. **Error and anomaly detection**: only partially suitable for unitless data; the physical units checking portion of this technique is only useful in code in which data with different units are being manipulated. Use of an automated tool is recommended.

f. **Structure analysis**: appears universally suitable for high level languages. It also appears to have little added value over error and anomaly detection for unit level testing, although at the CSC level its additional value is very apparent. Use of an automated tool is recommended.

203.3.4.2 Costs

The goal is to assemble the most effective testing techniques appropriate to the current software at the least cost. Cost can be related to effort, by multiplying the tester effort by the cost of any one tester's time. Cost analysis should include:

a. The cost of purchasing and maintaining the tools.

b. The cost of applying the techniques with the tools available in the development environment.

203.4 Detail to be Specified by Procuring Authority

a. Identify Path 3 for Selection Process (Task 203)

TASK SECTION 300

SOFTWARE RELIABILITY ESTIMATION

TASK SECTION 300
S/W RELIABILITY ESTIMATION & TESTING
OVERVIEW OF METHODOLOGY

300.1 Purpose

The purpose of Task 300 is to describe the general procedures for selecting and applying state-of-the-art software testing techniques, and for estimating software reliability during testing.

300.2 Documents Referenced in Task Section 300

MIL-STD-2167A
RADC TR-83-11
RADC TR-84-53
MIL-STD-756B
MIL-STD-785B

300.3 General Procedures

Perform software testing and make the Reliability Estimation. Use the measurements in Task 301 and 302 to estimate reliability.

300.3.1 Reliability Model

The general block diagrams applicable to software reliability prediction can also be used for software reliability estimation. For each block in the diagram (at a level where a block is a processing component like a computer), software reliability estimation is going to be based on performance results during test conditions.

Once software is executing its failure rate can be directly observed and a transformation is no longer required.

The failure rate of a program during test is expected to be affected by the amount of testing performed, the methodology employed, and the thoroughness of the testing. The following models are applicable to an estimation of the failure rate based on results from the test environment.

Estimating software reliability for a software component (whether it is at a system level where all software operates on one CPU or at a CSCI level with CSCIs operating on various CPUs) can be approached in two ways. The two approaches are described in the following paragraphs. Each requires observing the failure rate and testing time. Data collection procedures 12, 13, and 14 are used for measuring the software failure rate.

300.3.1.1 Reliability Estimation Modeling Approach 1

Several models have been suggested for relating failure experience to execution time (see RADC TR 83-11). The Musa Model as an example, assumes that the failure rate is proportional to the number of faults in a segment, and that the number of faults is being reduced every time a failure is encountered (not necessarily one fault removed for every failure encountered). This leads to an exponential distribution of faults with execution time, a one parameter distribution in which the scale parameter can be estimated by established methods. The Musa model has been shown to yield acceptable results for the test and early operational phases. The general prediction and estimation methodology can be used with any other execution time based model (RADC TR 83-11).

The failure rate during test, F, is given by

$$F = Lo \exp(- L1 * t)$$

where the amount of test time, t, is measured in terms of CPU-time, based on a 32-bit, 10 MIPS execution. Lo and L1 are the scale parameters proportional to the fault density. Any of the models described in RADC TR 83-11 can be used to model the failure rate observed during testing. Once modeled, the time until an acceptable failure rate is achieved can be calculated and operational performance can be estimated.

300.3.1.2 Reliability Estimation Modeling Approach 2

This approach does not use the models described in 300.3.1. It uses the failure rate observed during testing and modifies that rate by parameters estimating the thoroughness of testing and the extent to which the test environment simulates the operational environment.

The estimated failure rate then is:

$$F = FT1 * T1 \text{ or}$$

$$FT2 * T2$$

where FT1 is the average observed failure rate during testing,
and FT2 is the observed failure rate at end of test.

$$T1 = .02 * T$$

$$T2 = .14 * T$$

$$\text{and } T = TE * TM * TC$$

where TE is a measure of test effort, TM is a measure of test methodology and TC is a measure of test coverage. Definitions and calculation of these measures is in Task 301.

The most significant aspect of the test environment is that it represents a deliberate increase in the potential for detecting failure by:

a. Construction of test cases that represent a much higher variability of the input and control states than is expected in operation;

b. Close scrutiny of the computer output so that practically all failures that do occur are detected; and

c. Creating a high workload, particularly for stress tests, which increases the probability of failure.

Empirical data has shown the the average rate during test is 50 times greater than during operation and the failure rate at end of test is approximately 7 times that observed during operation. The .02 term in the T1 equation and the .14 term in the T2 equation above represent these observations.

The stress the operating environment will have on the software also must be taken into account. The basic failure rate relation for the initial operating environment is similar to that developed for the test environment except that the operating environment metric, E, replaces the test environment factor.

$$F = FT2 * T2 * E$$

Here T2 is .14 and the baseline value for E is 1. Modifiers for the operating environment factor arise from variability of the data and control states (EV) and from workload (EW) as discussed in Task 302.

300.4 Details to be specified by Procuring Activity

a. Requirement of Tasks 301 and 302.

b. Definition of test phases to be used.

c. Definition of qualification test requirements.

d. Statement of requirement of discrepancy reporting.

300.5 Overview of the Process

Software reliability estimation is based on performance results during test conditions. Once software is executing its failure rate can be directly observed and a transformation is no longer required. The failure rate of a program during test is expected to be affected by the amount of testing performed, the methodology employed, and the thoroughness of the testing.

300.5.1 Reliability Estimation Computations

Reliability estimation is based on performance results during test conditions. Once software is executing its failure rate can be directly observed and a transformation is no longer required. The failure rate of a program during test is expected to be affected by the amount of testing performed, the methodology employed, and the thoroughness of the testing.

300.5.2 Reliability Estimation Number for Test Environments

The Reliability Estimation Number (REN) is an Estimated Failure Rate (F). REN Model 2 is specified in Task 301 and provides the basis for estimating software reliability for test environments. It uses the failure rate observed during testing and modifies that rate by parameters estimating the thoroughness of testing and the extent to which the test environment simulates the operational environment. Tables TS300-1 and TS300-2 identify these REN data elements and procedures and their respective data collection sources and metric worksheets.

There are two RENs which can be computed for the dynamic test techniques. They are referred to as REN_AVG (average failure rate during test) and REN_EOT (failure rate at end of test). Computation of these RENs is presently infeasible for the static test techniques.

TABLE TS300-1 RED DATA COLLECTION PROCEDURES

METRIC DATA (TASK 301)	DATA COLLECTION PROCEDURE
Average Failure Rate During Test (FT1)	12, 13, 14
Failure Rate at End of Test (FT2)	12, 13, 14
Test Effort (TE)	15
Test Method (TM)	16
Test Coverage (TC)	17

TABLE TS300-2 REN DATA SOURCES

METRIC DATA	INPUT DOCUMENTS	METRIC WORKSHEETS
FT1	SPRs OS Reports Tester Logs	5, 6
FT2	SPRs OS Reports Tester Logs	5, 6
TE	Tester Logs	6
TM	Test Plans Test Procedures Software Development Plan Software Test Handbook	7
TC	Source Code Test Plans Test Procedures Requirements Document	8

REN_AVG and REN_EOT, also referred to as estimated failure rates (F), are computed as follows:

REN_AVG = F = FT1 * T1 or
REN_EOT = F = FT2 * T2

where: FT1 is the average observed failure rate during testing.
FT2 is the observed failure rate at end of test.

T1 = .02 * T
T2 = .14 * T and
T = TE * TM * TC

where: TE is a measure of Test Effort
TM is a measure of Test Methodology
TC is a measure of Test Coverage

The influence the test environment has on the estimated failure Rate (F) is described by three metrics. These metrics are in the form of a multiplier. The product of all of these metrics is used to adjust the Observed Failure Rate (FT) up or down depending on the level of confidence in the representativeness and thoroughness of the test environment.

300.5.2.1 Average Failure Rate During Testing (FT1)

FT1 can be calculated at any time during testing. It is based on the current total number of SPRs recorded and the current total amount of test operation time expended. It is expected that the failure rate will vary widely depending on when it is computed. For more consistent results, average failure rates are calculated for each test phase.

300.5.2.2 Failure Rate at End of Testing (FT2)

FT2 is based on the number of SPRs recorded and amount of computer operation time expended during the last three test periods of testing.

300.5.2.3 Test Effort (TE)

Three alternatives are provided for measurings TE and are based upon data availability:

a. The preferred alternative is based on the labor hours expended on software test. As a baseline, 40 percent of the total software development effort should be allocated to software test. A higher percentage indicates correspondingly more intensive testing, a lower percentage less intensive testing.

b. The second alternative utilizes funding instead of labor hours.

c. The third alternative is the total calendar time devoted to test.

Calculate TE, based on these three characteristics, as follows:

TE = .9 if 40/AT < 1, or
TE = 1.0 if 40/AT > 1

where: AT = the percent of the development effort devoted to testing.

300.5.2.4 Test Methodology (TM)

TM represents the use of test tools and test techniques by a staff of specialists. Task 301 specifies a technique to determine what tools and techniques should be applied to a specific application. That technique results in a recommended set of testing techniques and tools. The approach is to use that recommendation to evaluate the techniques and tools applied on a particular development.

Calculate TM as follows:

$TM = .9$ for $TU/TT > .75$
$TM = 1$ for $.75 > TU/TT > .5$
$TM = 1.1$ for $TU/TT < .5$

where: TU is the number of tools and techniques used.
TT is the number of tools and techniques recommended.

300.5.2.5 Test Coverage (TC)

TC assesses how thoroughly the software has been exercised during testing. If all of the code has been exercised then there is some level of confidence established that the code will operate reliably during operation. Typically however, these programs do not maintain this type of information and a significant portion (up to 40%) of the software (especially error handling code) may never be tested.

Calculate TC as follows:

$TC = 1/VS$

where: $VS = VS1$ during unit testing
$VS = VS2$ during CSC integration and test, and
$VS1 = (PT/TP + IT/TI)/2$

where: PT = execution branches tested
TP = total execution branches
IT = input tested
TI = total number of inputs

$VS2 = (MT/NM = CT/TC)/2$

where: MT = units tested
NM = total number of units
CT = interfaces tested
TC = total number of interfaces

300.5.3 REN for Operating Environments (E)

Task 302 provides the basis for estimating software reliability for operating environments. Several characteristics of the operational environment, experienced during OT&E, should be accounted for in estimating reliability. Again, during OT&E we are trying to extrapolate the observed failure rate (F) into operations. The characteristics we want to account for are the workload and the variability of inputs. These two characteristics, for which we have developed metrics, represent the stress of the operational environment on the software. The metrics will be multipliers which will raise or lower the estimated failure rate depending on the degree of stress ($E = EW * EV$).

a. Workload (EW)

The relationship between the workload and software failure rate has been investigated at Stanford University and a very significant positive correlation has been reported [ROSS82]. The basic concept underlying this phenomena is that more unusual situations (program swapped in and out of memory, queued I/O, wait states, etc.) are encountered in a heavy workload, and the application programmer may not have anticipated all the situations. In addition, system software will tend to fail more often when used more often.

Calculate EW as follows:

$$EW = ET/(ET-OS)$$

where: ET = total Execution Time
OS = Operating System overhead time

The use of operating system overhead was chosen because it is usually available. Other alternatives are number of system calls per minute, number of paging requests, and number of I/O operations.

b. Variability of Input (EV)

Variability of the input is the primary determinant of software reliability in some models. The basic concept here is that the greater the variability of inputs to the program the more likely an unanticipated input will be encountered and the program will fail.

The frequency of exception conditions can be used as a practical measure of variability. The monitoring of exception conditions is accomplished by hardware provisions which are incorporated in many current computers.

Calculate EV as follows:

$$EV = .1 + 4.5EC$$

where: EC = the number of Exception Conditions

300.5.4 Data Collection Tasks

This volume of the Guidebook contains Tasks 301 and 302 for Software Reliability Estimation and Testing. Each task contains data collection instructions and worksheets appropriate to the software test phase of the life cycle as follows.

a. Task 301: During testing and at end of testing.
b. Task 302: During system operation.

300.5.5 Data Collection Procedures

Each Task Section refers to data collection procedures. These procedures describe what data must be collected to use the software reliability estimation and testing computations described in Sections 300.5.1 and 300.5.2. Complementing these procedures are the instructions and actual worksheets. The intended process then is for reliability and test engineers to use the worksheets in conjunction with these data collection procedures to collect data. That data will then be used when the engineer or analyst uses the estimation algorithms to determine a reliability number. A data collection procedure index for Tasks 301 through 302 is provided in Table TS300-3. The procedures and worksheets are in the appendices.

TABLE TS300-3 TASK DATA COLLECTION PROCEDURE INDEX

TASK NO.	PROCEDURE NAME	PROCEDURE NO.
301	Fault Density	7
	Discrepancy Reports (DR)	12
	Execution Time (ET)	13
	Failure Rate (F)	14
	Test Effort (TE)	15
	Test Methodology (TM)	16
	Test Coverage (TC)	17
302	Exception Frequency (EV)	18
	Workload (EW)	19

TASK SECTION 301

RELIABILITY ESTIMATION FOR TEST ENVIRONMENT

TASK SECTION 301

RELIABILITY ESTIMATION FOR TEST ENVIRONMENT

301.1 Purpose

The purpose of Task 301 is to describe the procedures for estimating what the operational reliability will be based on observed failure rate during testing.

301.2 Documents Referenced in Task 301

See Task Section 300.

301.3 General Procedures

The influence the test environment has on the estimate of failure rate is described by three parameters as described in the following paragraphs.

Several characteristics of the test environment should be accounted for in the estimation of reliability. The observed failure rate may not accurately represent what the operational reliability will be because:

a. The test environment does not accurately represent the operational environment,

b. The test data does not thoroughly exercise the system thereby leaving untested many segments of the code,

c. The testing techniques employed do not thoroughly test the system, and

d. The amount of testing time does not thoroughly test the system.

These characteristics are taken into account by the metrics to be discussed in this paragraph. In each case the metrics will be in the form of a multiplier, the product of all of these to be used to adjust the observed failure rate (F_T) up or down depending on the level of confidence in the representativeness and thoroughness of the test environment.

Determination of Failure Rate During Test

Using Data Collection Procedures 7, 12 and 13 and Metric Worksheets 12E and 13E, calculate the current average failure rate during testing (FT1). The average current average failure rate during testing (FT1). The average failure rate during testing can be calculated at anytime during formal testing. It is based on the current total number of discrepancy reports recorded and the current total amount of test operation time expended. It is expected that the failure rate will vary widely depending on when it is computed. For more consistent results, average failure rates should be calculated for each software test phase: CSC Integration and testing, CSCI Testing; and, if required, for each system test phase: Systems Integration and Testing, and Operational Testing and Evaluation.

If the estimation is being made at the end of testing prior to deployment of the system, the estimation can be based on the failure rate observed at the end of CSCI testing (FT2). The failure rate calculation in this case is based on the number of discrepancy reports recorded and amount of computer operation time expended during the last three test periods of CSCI testing. Data Collection Procedure 14 should be used to calculate FT1 and FT2.

Estimate Software Reliability

Using the currently observed average failure rate during testing, an estimate of the operational failure rate can be calculated by:

$$F = FT1 * T1$$

$$\text{where } T1 = .02 *TE*TM*TC$$

The multipliers TE, TM and TC are determined as follows:

Test Effort (TE) - Optional

a. Three alternatives are provided for measuring test effort. The choice will primarily depend on availability of data. Data Collection Procedure 15 and Worksheet 14E aid in the collection and calculation of this metric.

1. The first alternative is based on the test budget. As a baseline 40% of the total development budget should be allocated to test. A higher percentage indicates correspondingly more intensive testing, a lower percentage less intensive testing.

2. The second alternative utilizes labor hours instead of budget.

3. The third alternative is the total calendar time devoted to test. The baseline should be total calendar time for a project of the same size.

b. The metric, TE, will be set based on observing these three characteristic during the validation phase of the project. Use Data Collection Procedure 15. The three characteristics impact TE as follows:

$$\text{if } 40/AT \leq 1$$

where AT = the percent of the development effort devoted to testing, then TE = .9

$$\text{or if } 40/AT \geq 1$$

where AT = the percent of the development schedule devoted to testing, then set TE = 1.0.

Test Methodology (TM) - Optional

a. The test methodology factor, TM, represents the use of test tools, and test techniques. In most cases the tools, and techniques are being operated by a staff of specialists who are also aware of other advances in software test technology.

b. Alternate methods to determine what tools and techniques should be applied to a specific application are provided in Task Section 200, and result in a recommended set of testing techniques and tools. The approach is to use that recommendation to evaluate the techniques and tools applied on a particular development. Use Procedure 15 and

Worksheet 15E. This evaluation will result in a score that will be the basis for this metric as follows:

$TM = .9$ for $TU/TT > .75$
$TM = 1$ for $.75 \geq TU/TT \geq .5$
$TM = 1.1$ for $TU/TT < .5$

where TU is the number of tools and techniques used and TT is the number recommended.

Test Coverage (TC) - Recommended

a. This metric assesses how thoroughly the software has been exercised during testing. If all of the code has been exercised then there is some level of confidence established that the code will operate reliably during operation. Typically however, test programs do not maintain this type of information and a significant portion (up to 40%) of the software (especially error handling code) may never be tested. Use data collection procedure 16 and Data Collection Worksheet 17E.

b. This metric can be calculated in three ways depending on the phase of testing as follows:

$$TC = 1/VS$$

where VS = VS1 during unit testing
= VS2 during integration testing
= VS3 during system testing

and $VS1 = (PT/TP + IT/TI)/2$
where PT = execution branches tested
TP = total execution branches
IT = input tested
TI = total number of inputs
$VS2 = (MT/TM + CT/TC)/2$
where MT = units tested
TM = total number of units
CT = interfaces tested
TC = total number of interfaces
$VS3 = RT/NR$
where RT = Requirements tested
NR = total number of requirements

An updated reliability estimation can be made using these multipliers at the end of test by using:

$$F = FT2 * T2$$

where $T2 = .14 * TE*TM*TC$

A comparison of the predicted fault density (determined using Tasks 101 through 104 in Volume 3) with the actual fault density realized can be made. Using Data Collection procedure 16E, the fault density realized is the number of discrepancy reports reported during testing divided by the total number of lines of code in the system. A comparison of the predicted failure rate, transformed from the predicted fault density, can also be made with the estimated failure rate calculated in this task. S ;nificant variation in these values suggests that analyses be conducted to evaluate the differences. Consistent values suggests accurate predictions and estimations.

301.4 Detailed to be Specified by the Procuring Authority

a. Define the software component level for estimation (different levels may be specified for each life cycle phase)

b. Define life cycle phases to be covered and estimation milestones.

c. Identify data collection procedures (see Appendix B of this Guidebook)

301.5 Procedures

The applicable procedures to this Task are:

Procedure No.	Title
6	Fault Density (See Task Section 100)
7	Discrepancy Reports (DR) (See Task Section 100)
12	Execution Time (ET)
13	Failure Rate (F)
14	Test Effort (TE)
15	Test Methodology (TM)
16	Test Coverage (TC)

TASK SECTION 302

SOFTWARE RELIABILITY ESTIMATION FOR OPERATING ENVIRONMENT

TASK SECTION 302

SOFTWARE RELIABILITY ESTIMATION FOR OPERATING ENVIRONMENT

302.1 Purpose

The purpose of task 302 is to describe the procedures for estimating what the operational reliability will be based on estimates of the operational environment and the observed failure rate at the end of test.

302.2 Documents Referenced in Task Section 302

See Task Section 300.

302.3 General Procedures

Two factors are accounted for in estimating the failure rate for the operational environment: the workload expected and the input variability. These both represent expected stress on the system.

Estimate Software Reliability

Using the end of test failure rate (see Task 301 and Data Collection Procedure 13), FT2, an estimate of the operational reliability is calculated as follows:

$F = F_{T2} * T_2*E$

where F_{T2} is the failure rate at end of test

$T_2 = .14$

$E = EV*EW$, modifiers representing stress of input variability, EV, and workload, EW.

The modifiers are calculated as follows:

Variability of Data and Control States (EV) - Recommended

a. Software that is delivered for Air Force use will be essentially fault free for nominal data and control states, i.e., where an input is called for, an input fully compliant with the specification will be present; when an output is called for, the channel for receiving the output will be available. A major factor in the occurrence of failures, and therefore affecting the failure rate, will be the variability of input and control states.

b. The frequency of exception conditions as a measure of variability is used here. Exception states include:

   1. Page faults, input/output operations, waiting for completion of a related operation -- the frequency of all of these is workload dependent and the effect on software reliability is discussed in the next section;

   2. Response to software deficiencies such as overflow, zero denominator, or array index out of range; and

3. Response to hardware difficulties such as parity errors, error correction by means of code, or noisy channel.

The last two of these are combined in the input variability modifier for the operating environment, EV. Data illustrated in Table TS202-1, indicates that approximately 1,000 exception conditions of the latter two types were encountered in 5,000 hours of computer operation. A value of 0.2 exception conditions per computer hour has therefore been adopted as the baseline, to be equated to unity. Because failures may arise even if no exception conditions at all are encountered, it is desirable to bias the modifier to a small positive value. The suggested form is

$$EV = 0.1 + 4.5EC$$

where EC is the number of exception conditions per hour. For E = 0.2, EV = 1. Use data collection procedure 17.

Workload (EW) - Recommended

Significant effects of workload on software failure rate have been reported. The hazard, the incremental failure rate due to increasing workload, ranges over two orders of magnitude.

For military applications, workload effects can be particularly important. During time of conflict, the workloads can be expected to be exceptionally heavy, causing the expected failure rate to increase, and yet at that same time a failure can have the most serious consequences. Hence, predictions of failure rates that do not take workload effects into account fail to provide the information that Air Force decision makers need.

The mechanism by which workload increases the failure rate is not completely known, but it is generally believed to be associated with a high level of exception states, such as busy I/O channels, long waits for disk access, and possibly increased memory errors (due to the use of less frequently accessed memory blocks). Data show that the highest software (and also hardware) failure rates were experienced during the hours when the highest levels of exception handling prevailed.

Details of workload effects on software failure rate are still a research topic, and no specific work in that area has been included in this Guidebook. The estimations will be based on published work, such as Figure TS302-1. The quantity plotted along the vertical axis is the inherent load hazard, z(x), defined as:

Probability of failure in workload interval (x, x+ delta x)
Probability no failure in interval (0,x).

It measures the incremental risk of failure involved in increasing the workload from x to x+delta x.

The horizontal axis shows three different measures of workload:

a. Virtual memory paging activity, number of pages read per second (PAGEIN);

b. Operating system overhead, fraction of time not available for user processes (OVERHEAD); and

c. Input/output activity, number of non-spooled input/ output operations started per second (SIO).

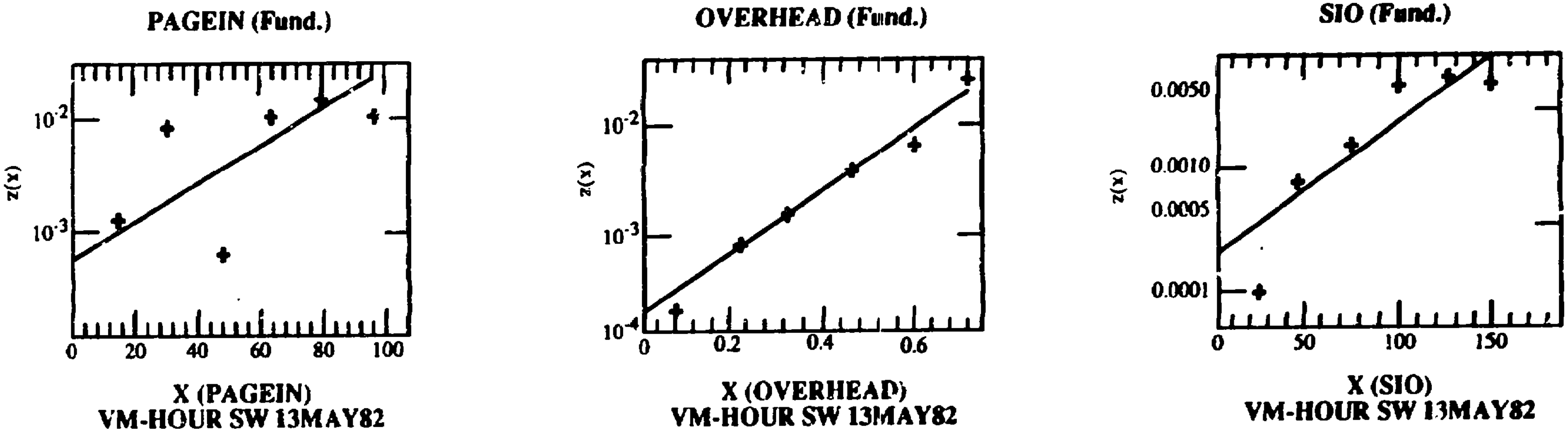


FIGURE TS302-1 EFFECT OF WORKLOAD ON SOFTWARE HAZARD

These graphs provide an option of estimating workload effects by any of the indicators of workload used here. The fraction of overhead usage is probably the most commonly obtainable quantity. From a practical point of view, before a computer installation becomes operational, the fraction of capacity to be used at maximum expected workload is probably the only indication of this factor that will be available early in the development. Data Collection Procedure 18 and Worksheet 18E should be referenced.

The workload metric takes the form

$$EW = ET/(ET-OS)$$

where ET = Total Execution Time
OS = Operating System overhead time

302.4 Detail to be Specified by the Procuring Authority

a. Define the software component level for estimation (different levels may be specified for each life cycle phase)

b. Define life cycle phases to be covered and estimation milestones

c. Identify data collection procedures.

302.5 Procedures, Instructions and Worksheets

Procedure No.	Title
17	Exception Frequency (EV)
18	Workload (EW)

APPENDIX A

DEFINITIONS AND TERMINOLOGY

APPENDIX A

DEFINITIONS AND TERMINOLOGY

This appendix presents definitions of the principal terms and concepts used in this report. Where possible, the definitions are taken from established dictionaries or from the technical literature. Where a rationale for the selection or formulation of a definition seems desirable, it is provided in an indented paragraph following the definition. The sources for the definitions will be found in the list of references at the end of this Guidebook.

ERROR - A discrepancy between a computed observed, or measured value or condition and the true, specified, or theoretically correct value or condition. [ANSI81]

This definition is listed as (1) in the American National Dictionary for Information Systems. Entry (2) in the same reference states that error is a "Deprecated term for mistake". This is in consonance with [IEEE83] which lists the adopted definition as (1) and lists as (2) "Human action that results in software containing a fault. Examples include omission or misinterpretation of user requirements in a software specification, incorrect translation or omission of a requirement in a design specification. This is not a preferred usage."

FAILURE - The inability of a system or system component to perform a required function with specified limits. A failure may be produced when a fault is encountered. [IEEE83]

This definition is listed as (2) in the cited reference which lists as (1) "The termination of the ability of a functional unit to perform its required function" and as (3) "A departure of program operation from program requirements". Definition (1) is not really applicable to software failures because these may render an incorrect value on one iteration but correct values on subsequent ones. Thus, there is no termination of the function in case of a failure. Definition (3) was considered undesirable because it is specific to the operation of a computer program and a more system-oriented terminology is desired for the purposes of this study.

FAULT - An accidental condition that causes a functional unit to fail to perform its required function. [IEEE83]

This definition is listed as (1) in the cited reference which lists as (2) "The manifestation of an error (2) in software. A fault, if encountered, may cause a failure". Error (2) is identified a synonymous with "mistake". Thus this definition states that a fault is the manifestation in software of a (human) mistake. This seems less relevant than the primary definition. It is recognized that the presence of a fault will not always or consistently cause a unit to fail since the presence of a specific environment and data set may also be required (see definition of software reliability).

MISTAKE - A human action that produces an unintended result. [ANSI81]

SOFTWARE QUALITY FACTOR - A broad attribute of software that indicates its value to the user, in the present context equated to reliability. Examples of software quality factors are maintainability, portability, as well as reliability. May also be referred to simply as factor or quality factor. [Based on MCCA80]

SOFTWARE QUALITY METRIC - A numerical or logical quantity that measures the presence of a given quality factor in a design or code. An example is the measurement of size in terms of lines of executable code (a quality metric). May also be referred to simply as metric or quality metric. A single quality factor may have more than one metric associated with it. A metric typically is associated with only a single factor. [Based on MCCA80]

SOFTWARE RELIABILITY - The probability that software will not cause the failure of a system for a specified time under specified conditions. the probability is a function of the inputs to and use of the system as well as a function of the existence of faults in the software. The inputs to the system determine whether existing faults, if any, are encountered. [IEEE83]

This definition is listed as (1) in the IEEE Standard Glossary. An alternate definition, listed as (2), is "The ability of a program to perform a required function under sated conditions for a specified period of time." This definition is not believed to be useful for the current investigation because (a) it is not expressed as a probability and therefore cannot be combined with hardware reliability measures to form a system reliability measure, and (b) it is difficult to evaluate in an objective manner. The selected definition fits well with the methodology for software reliability studies which will be followed in this study, particularly in that it emphasizes that the presence of faults in the software as well as the inputs and conditions of use will affect reliability.

SOFTWARE RELIABILITY MEASUREMENT - The life-cycle process of establishing quantitative reliability goals, predicting, measuring and assessing the progress and achievement of those goals during the development, testing, and O&M phases of a software system.

SOFTWARE RELIABILITY PREDICTION - A numerical statement about the reliability of a computer program based on characteristics of the design or code, such as number of statements, source language or complexity [HECH77]

Software reliability prediction is possible very early in the development cycle before executable code exists. The numeric chosen for software reliability prediction should be compatible with that intended to be used in estimation and measurement.

SOFTWARE RELIABILITY ESTIMATION - The interpretation of the reliability measurement on an existing program (in its present environment, e.g., test) to represent its reliability in a different environment (e.g., a later test phase or the operations phase). Estimation requires a quantifiable relationship between the measurement environment and the target environment. [HECH77]

The numeric chosen for estimation must be consistent with that used in measurement.

SOFTWARE RELIABILITY ASSESSMENT - Generation of a single numeric for software reliability derived from observations on program execution over a specified period of time. Defined sections of the execution will be scored as success or failure. Typically, the software will not be modified during the period of measurement, and the reliability numeric is applicable to the measurement period and the existing software configuration only. [HECH77]

The statement about not modifying the software during the period of measurement is necessary in order to avoid committing to a specific model of the debugging/reliability relation. In practice, if the measurement interval only a small fraction of the existing faults are removed, then the occurrence of modifications will not materially affect the measurement.

PREDICTIVE SOFTWARE REALIBILITY FIGURE-OF-MERIT (RP) - A reliability number (fault density) based on characteristics of the application, development environment, and software implementation. The RP is established as a baseline as early as the concept of the system is determined. It is then refined based on how the design and implementation of the system evolves.

RELIABILITY ESTIMATION NUMBER (RE) - A reliability number (failure rate) based on observed performance during test conditions.

FUNCTION - A specific purpose of an entity or its characteristic action. [ANSI81] A subprogram that is invoked during the evaluation of an expression in which its name appears and that returns a value to the point of invocation. Contrast with subroutine. [IEEE83]

MODULE - A program unit that is discrete and identifiable with respect to compiling, combining with other units, and loading; for example, the input to, or output from, an assembler, compiler, linkage editor, or executive routine. [ANSI81] A logically separable part of a program. [IEEE83]

SUBSYSTEM - A group of assemblies or components or both combined to perform a single function. [ANSI73] In our context, a subsystem is a group of modules interrelated by a common function or set of functions. Typically identified as a Computer Program Configuration Item (CPCI) or Computer Software Configuration Item (CSCI). A collection of people, machines, and methods organized to accomplish a set of specific functions. [IEEE83] An integrated whole that is composed of diverse, interacting, specialized structures and subfunctions. [IEEE83] A group or subsystem united by some interaction and interdependence, performing many duties but functioning as a single unit. [ANSI73]

SYSTEM - In our context, a software system is the entire collection of software modules which make up an application or distinct capability. Along with the computer hardware, other equipment (such as weapon or radar components), people and methods the software system comprises an overall system.

APPENDIX B

PROCEDURES AND WORKSHEETS

PROCEDURE NO. 0

1. Title: Application Type (A)

2. Prediction or Estimation Parameter Supported: Application Type (A)

3. Objectives: At the system level categorize the system application according to the application and time dependence schemes identified in Worksheet 0.

4. Overview: Manual inspection of documentation to determine the type of system according to preceding classifications. This determination can be made at the Concept Definition phase.

5. Assumptions/Constraints: Ambiguities or other difficulties in applying this scheme should be resolved in favor of the dominant or most likely classification.

6. Limitations: None

7. Applicability: Identify Application Type at project initiation. Metric worksheets require update of information at each major review. It should not change.

8. Required Inputs: Statement of Need (SON), Required Operational Capability (ROC), or system requirements statement should indicate application type.

9. Required Tools: Visual inspection of documentation.

10. Data Collection Procedures: Functional description of system extracted from documentation and matched with an application area.

11. Outputs: A baseline fault density, A, will be associated with each Application Type.

12. Interpretation of Results: Application type may be used early in the development cycle to predict a baseline fault density. These rates are then modified as additional information concerning the software becomes available.

13. Reporting: Application type, together with projected baseline fault density, is reported. The baseline rate should be made available to the prospective user to ensure that the user is aware of failure rates (or fault density) for this application and has provisions which will affect the characteristics of the specific software as they unfold during system development.

14. Forms: Use Metric Worksheet 0.

15. Instructions: Perform the following steps using **Worksheet 0** and record data for determination of Application RPFOM for the system.

    Step 1. Review pertinent documentation as needed (Table TS100-3)
    Step 2. Complete header information on answer sheet.
    Step 3. Select name of one of the six Applications listed on worksheet.
    Step 4. Record current data and Application name under **Item I** on answer sheet.

16. Potential Plans for Automation: Information for this factor will be obtained manually. The Prototype IRMS may be used to automate the calculation.

METRIC WORKSHEET 0
SYSTEM SOFTWARE DEFINITION
SYSTEM LEVEL

GENERAL INFORMATION

1. PROJECT ______________________________

2. DATE ______________________________

3. ANALYST ______________________________

4. PRODUCT ______________________________

5. SOURCE DOCUMENTATION

____

____

____

____

____

METRIC WORKSHEET 0: APPLICATION TYPE
PHASE: Pre Software Development
APPLICATION LEVEL: System

APPLICATION TYPE	AVERAGE FAULT DENSITY
1. AIRBORNE SYSTEMS - Manned Spacecraft - Unmanned Spacecraft - Mil-Spec Avionics - Commercial Avionics	0.0128
2. STRATEGIC SYSTEMS - C3I - Strategic C2 Processing - Indications and Warning - Communications Processing	0.0092
3. TACTICAL SYSTEMS - Strategic C2 Processing - Communication Processing - Tactical C2 - Tactical MIS - Mobile - EW/ECCM	0.0078
4. PROCESS CONTROL SYSTEMS - Industrial Process Control	0.0018
5. PRODUCTION SYSTEMS - MIS - Decision Aids - Inventory Control - Scientific	0.0085
6. DEVELOPMENTAL SYSTEMS - Software Development Tools - Simulation - Test Beds - Training	0.0123

PROCEDURE NO. 1

1. Title: Development Environment (D)

2. Prediction or Estimation Parameter Supported: Development Environment (D)

3. Objectives: Categorizes the development environment according to Boehm's [BOEH81] classification. Additional distinguishing characteristics derived from RADC TR 85-47 are also used.

4. Overview: In Boehm's classification the system is categorized according to environment as follows:

   a. Organic Mode -- The software team is part of the organization served by the program.

   b. Semidetached Mode -- The software team is experienced in the application but not affiliated with the user.

   c. Embedded Mode -- Personnel operate within tight constraints. The team has much computer expertise, but is not necessarily very familiar with the application served by the program. System operates within strongly coupled complex of hardware, software, regulations, and operational procedures.

   A survey in RADC TR 85-47 revealed the following factors, were felt to have significant impact on the reliability of software. They, therefore, provide a worksheet for predicting the quality of software produced using them:

   a. Organizational Considerations

   b. Methods Used

   c. Documentation

   d. Tools Used

   e. Test Techniques Planned

The developmental environment should be described in the Software Development Plan, and the testing environment in the Software Test Plan. If it is not, it will be necessary to review product reports or to interview the software developers.

5. Assumptions/Constraints: Use of the Boehm metric assumes a single dimension along which software projects can be ordered, ranging from organic to embedded. Care must be taken to ensure that there is some allowance made for variations from this single-dimensional model -- e.g. when inexperienced personnel are working in an in-house environment. In such cases, the dominant or most important characteristic will be used.

   The worksheet developed from RADC TR 85-47 provides a rating for the developmental environment and process. Higher numbers of methods and tools planned for use are assumed to be associated with more reliable software. However, this relationship is not likely to be linear (that is, it is not likely that each item on the checklist will increase reliability by an identical amount). Calibration of the score will be required during tests of the metrics. Current values are from a survey.

6. Limitations: The reliability of these metrics will be affected by the subjective judgments of the person collecting the data. Data concerning project personnel may not always be available after project completion, unless it has been specifically gathered for this purpose.

7. Applicability: The Development Environment will be indicated during the requirements phase and, combined with expected fault density/failure rates for the Application Area, can be used to obtain an early forecast of reliability.

8. Required Inputs: Information is extracted visually from requirements or specifications documentation.

9. Required Tools: Manual data extraction from existing documentation. A checklist is provided in the Data Collection Worksheet 1.

10. Data Collection Procedures: Using the classification scheme and checklist in Metric Worksheet 1, use Software Development Plan to determine the Development Environment metric. Where appropriate information is not included in available documentation, it may be necessary to interview project personnel.

11. Outputs: Classification and completed checklist as indicated in paragraph 9 above (Metric inputs D_O and D_C).

12. Interpretation of Results: As a refinement, regression techniques can be used to obtain metric values for each of the indicated environments in the Boehm classification. These are combined with the score obtained from the worksheet to obtain the score for this factor.

13. Reporting: Where the predicted failure rate differs from specified or expected values, changes in the personnel mix, project organization, methodology employed, or other environmental factors may be required to improve predicted reliability or to reduce costs. Early reporting of this information will permit such changes to be made in a timely fashion.

14. Forms: Use Metric Worksheet 1

15. Instructions: Perform the following steps using **Worksheet 1A** and **Answer Sheet 1 (Appendix C)** to collect and record data for determining a first approximation of Development Environment RPFOM for the system:

    Step 1a. Review pertinent documentation as needed (Table TS100-3).

    Step 1b. Select name of one of the three Development Environment options listed on work sheet.

    Step 1c. Record current date and Development Environment name under **Item II** on answer sheet.

    Step 1d. **Item II** response can now be entered into automated database.

    Perform the following steps using **Worksheet 1B** and **Answer Sheet 1 (Appendix C)** to collect and record data for determining a second **Development Environment** RPFOM for the system:

    Step 2a. Review pertinent documentation as needed (Table TS100-3).

Step 2b. Record current date for **Item III** on answer sheet. and circle "Y" or "N" for items 1a through 4j based upon applicability to the system of Development Environment Characteristics listed in the worksheet (i.e., circle "Y" for characteristics that apply to the project development environment, "N" to characteristics that are not applicable).

16. Potential/Plans for Automation: Information for this factor will be obtained manually. The IRMS may be used to automate the calculation of this factor and a refined RPFOM.

METRIC WORKSHEET 1A: DEVELOPMENT ENVIRONMENT (1)
PHASE: Pre Software Development
APPLICATION LEVEL: System

DEVELOPMENT ENVIRONMENT	DESCRIPTION	METRIC
ORGANIC MODE	The software team is part of the organization served by the program.	0.76
SEMI-DETACHED MODE	The software team is experienced in the application but not affiliated with user.	1.0
EMBEDDED MODE	Personnel operate within tight constraints. The software team has much computer expertise, but may be unfamiliar with the application served by the program. System operates within strongly coupled complex of hardware, software regulations, and operational procedures.	1.3

METRIC WORK SHEET 1B: DEVELOPMENT ENVIRONMENT (2)
PHASE: Pre Software Development
APPLICATION LEVEL: System

1. ORGANIZATIONAL CONSIDERATIONS

   1a. Separate design and coding
   1b. Independent test organization
   1c. Independent quality assurance
   1d. Independent configuration management
   1e. Independent verification and validation
   1f. Programming team structure
   1g. Educational level of team members above average
   1h. Experience level of team members above average

2. METHODS USED

   2a. Definition/Enforcement of standards
   2b. Use of higher order language (HOL)
   2c. Formal reviews (PDR, CDR, etc.)
   2d. Frequent walkthroughs
   2e. Top-down and structured approaches
   2f. Unit development folders
   2g. Software Development library
   2h. Formal change and error reporting
   2i. Progress and status reporting

3. DOCUMENTATION

   3a. System Requirements Specification
   3b. Software Requirements Specification
   3c. Interface Design Specification
   3d. Software Design Specification
   3e. Test Plans, Procedures, and Reports
   3f. Software Development Plan
   3g. Software Quality Assurance Plan
   3h. Software Configuration Management Plan
   3i. Requirements Traceability Matrix
   3j. Version Description Document
   3k. Software Discrepancy

METRIC WORKSHEET 1B: DEVELOPMENT ENVIRONMENT (2) (Con't.)
PHASE: Pre Software Development
APPLICATION LEVEL: System

4. TOOLS USED

   4a. Requirements Specification Language
   4b. Program Design Language
   4c. Program Design Graphical Technique (flowchart, HIPO, etc.)
   4d. Simulation/Emulation
   4e. Configuration Management
   4f. Code Auditor
   4g. Data Flow Analyzer
   4h. Programmer Workbench
   4i. Measurement Tools

5. TEST TECHNIQUES PLANNED

   5a. Code Review
   5b. Branch Testing
   5c. Random Testing
   5d. Functional Testing
   5e. Error & Anomaly Detection
   5f. Structure Analysis

PROCEDURE NO. 2

1. Title: Anomaly Management (SA)

2. Prediction or Estimation Parameter Supported: Software Characteristics

3. Objectives: The purpose of this procedure is to determine the degree to which a software system is capable of responding appropriately to error conditions and other anomalies.

4. Overview: This metric is based on the following characteristics:

   a. Error Condition Control,

   b. Input Data Checking,

   c. Computational Failure identification and Recovery,

   d. Hardware Fault Identification and Recovery,

   e. Device Error Identification and Recovery, and

   f. Communication Failure Identification and Recovery.

   In general, it is assumed that the failure rate of a system will decrease as anomaly management, as measured by this metric, improves.

   This metric requires a review of program requirements, specifications, and designs to determine the extent to which the software will be capable of responding appropriately to non-normal conditions, such as faulty data, hardware failures, system overloads, and other anomalies. Mission-critical software should never cause mission failure. This metric determines whether error conditions are appropriately handled by the software, in such a way as to prevent unrecoverable system failures.

5. Assumptions/Constraints: Elements of this metric are obtained manually in checklist form. This metric assumes that system requirements and specifications contain sufficient information to support computation of the required values.

6. Limitations: By its very nature, an anomaly is an unforeseen event, which may not be detected by error-protection mechanisms in time to prevent system failure. The existence of extensive error-handling procedures will not guarantee against such failures, which may be detected during stress testing or initial trial implementation. However, the metric will assist in determining whether appropriate error procedures have been included in the system specifications and designs.

7. Applicability: Elements of this metric will be obtained throughout the software development cycle.

8. Required Inputs: This procedure requires a review of all system documentation and code.

9. Required Tools: No tools will be used in the collection of data for this metric. A checklist is provided in the Worksheets.

10. Data Collection Procedures: Data to support this metric will be collected during system development. Data must be obtained manually, through inspection of code and documentation.

11. Outputs: The measurement, AM, is the primary output of this procedure. In addition, reports of specific potential trouble areas, in the form of discrepancy reports, will be desirable for guidance of the project manager and the program supervisor.

12. Interpretation of Results: Anomaly conditions require special treatment by a software system. A high score for AM would indicate that the system will be able to survive error conditions without system failures.

13. Reporting: An overall report concerning anomaly management will be prepared. It should be noted that the cost of extensive error-handling procedures must he balanced against the potential damage to be caused by system failure. A proper balance of costs and benefits must be determined by project management; the purpose of this metric is to assist the manager in assessing these costs and benefits.

14. Forms: Metric Worksheet 2.

15. Instructions: The following worksheets are used to assess the degree to which anomaly management (error tolerance) is being built into a software system. The worksheets should be applied as follows:

WORKSHEET	APPLICATION
2A	During Software Requirements Analysis (at SSR)
2B	During Preliminary Design (at PDR)
2C/2D	During Detailed Design and Coding (at CDR)

Note: First, complete Worksheet 2. Then complete the remaining worksheets as follows. Calculate a value if required. Check Yes or No in response to the question. Check NA to a question that is not applicable and these do not count in calculation of metric. You may enter your answers directly on the worksheets or on the provided answer sheet.

Perform the following steps using **Worksheet 2A** and **Answer Sheet 2 (Appendix C)** to collect and record data for measuring Anomaly **Management at SSR** for each CSCI of the system:

Step 1a. Review pertinent documentation as needed (Table TS100-3).

Step 1b. Record header information on answer sheet.

Step 1c. Record current date for **Item I** on answer sheet, and complete items AM.1(1) through RE.1(4) based on questions in worksheet.

Perform the following steps using **Worksheet 2B** and **Answer Sheet 3 (Appendix C)** to collect and record data for measuring **Anomaly Management** at **PDR** for each CSCI of the system:

Step 4a. Review pertinent documentation as needed (Table TS100-3).

Step 4b. Record header information on answer sheet.

Step 4c. Record current date for **Item I** on answer sheet, and complete items AM.3(1) through RE.1(4) based on questions in worksheet.

Perform the following steps using **Worksheet 2C** and **Answer Sheet 4 (Appendix C)** to collect and record data for performing **Anomaly Management** at **CDR** for each Unit of the CSCI:

Step 7a. Review pertinent documentation as needed (Table TS100-3).

Step 7b. Record header information on answer sheet.

Step 7c. Record current date for **Item I** on answer sheet, and complete items AM.1(3) through AM.2(7) based on questions in worksheet.

Perform the following steps using **Worksheet 2D** and **Answer Sheet 5 (Appendix C)** to collect and record data for measuring **Anomaly Management** at **CDR** for each CSCI of the system:

Step 9a. Review pertinent documentation as needed (Table TS100-3).

Step 9b. Record header information on answer sheet.

Step 9c. Record current date for **Item I** on answer sheet, and complete items AM.1(3) through AM.3(4) based on questions in worksheet.

16. Potential/Plans for Automation: Information for this metric is obtained manually.The IRMS may be used to automate the calculation.

17. Remarks: Proper determination of this metric will require some imagination and intelligent judgment on the part of the reviewer. Since error conditions take a wide variety of forms, the reviewer should be experienced in developing error-resistant software.

METRIC WORKSHEET 2
ANOMALY MANAGEMENT

GENERAL INFORMATION

1. PROJECT ____

2. DATE ____

3. ANALYST ____

4. PRODUCT ____

5. SOURCE DOCUMENTATION

____

____

____

____

____

6. PHASE	SRR	______
	PDR	______
	CDR	______
	CODING	______

(Check applicable one)

7. LEVEL:	System	______		
	CSCI	______	NAME	______
	CSC	______	NAME	______
	UNIT	______	NAME	______

(Complete applicable level)

METRIC WORKSHEET 2A: ANOMALY MANAGEMENT (1)
PHASE/REVIEW: Software Requirements Analysis/SSR
APPLICATION LEVEL: CSCI

		Yes	No	NA	Unk
AM.1(1)	a. How many instances are there of different processes (or functions, subfunctions) which are required to be executed at the same time (ie., concurrent processing)? b. How many instances of concurrent processing are required to be centrally controlled? c. Calculate b/a. d. If b/a < 1, Circle N. If ba = 1, Circle Y.				
AM.1(2)	a. How many error conditions are required to be recognized (identified)? b. How many recognized error conditions require recover or repair? c. Calculate b/a. d. If b/a < 1, Circle N. If b/a = 1, Circle Y.				
AM.1(3)	Is there a standard for handling recognized errors such that all error conditions are passed to the calling function or software element?				
AM.1(4)	a. How many instances exist of the same process (or function, subfunction) being required to execute more than once for comparison purposes (ie., polling of parallel or redundant processing results)? b. How many instances of parallel/redundant processing are required to be centrally controlled? c. Calculate b/a. d. If b/a < 1, Circle N. If b/a = 1, Circle Y.				
AM.2(1)	Are error tolerances specified for all applicable external input data (ie., range of numerical values, legal combinations of alphanumerical values)?				
AM.3(1)	Are there requirements for detection of and/or recovery from all computational failures?				
AM.3(2)	Are there requirements to range test all critical (ie., supporting a mission-critical function)loop and multiple transfer index parameters before use?				
AM.3(3)	Are there requirements to range test all critical (ie., supporting a mission-critical function) subscript values before use?				

METRIC WORKSHEET 2A: ANOMALY MANAGEMENT (1) (Con't.)
PHASE/REVIEW: Software Requirements Analysi 'SSR
APPLICATION LEVEL: CSCI

		Yes	No	NA	Unk
AM.3(4)	Are there requirements to range test all critical output data (ie., data supporting a mission-critical system function) before final outputting?				
AM.4(1)	Are there requirements for recovery from all detected hardware faults (ie., arithmetic faults, power failure, clock interrupt)?				
AM.5(1)	Are there requrements for recovery from all I/O divide errors?				
AM.6(1)	Are there requirements for recovery from all communication transmission errors?				
AM.7(1)	Are there requirements for recovery from all failures to communicate with other modes or other systems?				
AM.7(2)	Are there requirements to periodically check adjacent nodes or interoperating system for operational status?				
AM.7(3)	Are there requirements to provide a strategy for alternate routing of messages/				
RE.1(1)	Are there requirements to ensure communication paths to all remaining nodes/communication links in the event of a failure of one node/link?				
RE.1(2)	Are there requirements for maintaining the integrity of all data values following the occurrence of anomalous conditions?				
RE.1(2)	Are there requirements to enable all disconnected nodes to rejoin the network after recovery, such that the processing functions of the system are not interrupted?				
RE.1(4)	Are there requirements to replicate all critical data in the CSCI at two or more distinct nodes?				
AM SCORE:	Count the number of Y's checked. Count the number of N's checked. Calculate the number of N's divided by the number of N's and Y's. Assign that value to AM				
		AM=			

METRIC WORKSHEET 2B: ANOMALY MANAGEMENT (2)

PHASE/REVIEW: Preliminary Design/PDR
APPLICATION LEVEL: CSCI

		Yes	No	NA	Unk
AM.3(1)	Are there provisions for recovery from all computational failures?				
AM.4(1)	Are there provisions for recovery from all detected hardware faults (e.g., arithmetic faults, power failure, clock interrupt)?				
AM.5(1)	Are there provisions for recovery from all I/O device errors?				
AM.6(1)	Are there provisions for recovery from all communication transmission errors?				
AM.6(2)	Is error checking information (e.g., checksum, parity bit) computed and transmitted with all messages?				
AM.6(3)	Is error checking information computed and compared with all message receptions?				
AM.6(4)	Are transmission retries limited for all transmissions?				
AM.7(1)	Are there provisions for recovery from all failures to communicate with other nodes or other systems?				
AM.7(2)	Are there provisions to periodically check all adjacent nodes or interoperating systems for operational status?				
AM.7(3)	Are there provisions for alternate routing of messages?				
RE.1(1)	Do communication paths exist to all remaining nodes/links in the event of a failure of one node/link?				
RE.1(2)	Is the integrity of all data values maintained following the occurence of anomalous conditions?				
RE.1(3)	Can all disconnected nodes rejoin the network after recovery, such that the processing functions of the sytem are not interrupted?				
RE.1(4)	Are all critical data in the system (or CSCI) replicated at two or more distinct nodes, in accordance with specified requirements?				
AM SCORE:	Count the number of Y's circled and the number of N's circled. Calculate the ratio of the number of N's divided by the total number of N's and Y's. Assign that value to AM.				
	TOTALS				
		AM=			

METRIC WORKSHEET 2C: ANOMALY MANAGEMENT (3)

PHASE/REVIEW: Detailed Design/CDR
APPLICATION LEVEL: UNIT

		Yes	No	NA	Unk
AM.1(3)	When an error condition is detected, is resolution of the error determined by the calling unit?				
AM.2(7)	Is a check performed before processing begins to determine that all data is available?				
AM SCORE: Count the number of N's and divide by 2.					
	TOTALS				
		AM=			

METRIC WORKSHEET 2D: ANOMALY MANAGEMENT (4)

PHASE/REVIEW: Detailed Design/CDR
APPLICATION LEVEL: CSCI

		Yes	No	NA	Unk
AM.1(3)	a. How many units in CSCI?				
	b. For how many units, when an error condition is detected, is resolution of the error not determined by the calling unit?				
	c. Calculate b/a.				
	d. If b/a ≥ 0.5, circle N; otherwise, circle Y.				
AM.2(2)	Are values of all applicable external inputs with range specifications checked with respect to specified range prior to use?				
AM.2(3)	Are all applicable external inputs checked with respect to specified conflicting requests prior to use?				
AM.2(4)	Are all applicable external inputs checked with respect to specified illegal combinations prior to use?				
AM.2(5)	Are all applicable external inputs checked for reasonableness before processing begins?				
AM.2(6)	Are all detected errors, with respect to applicable external inputs, reported before processing begins?				
AM.2(7)	a. How many units in CSCI (see AM.1(3)a)?				
	b. How many units do not perform a check to determine that all data is available before processing begins?				
	b. Calculate b/a.				
	c. If b/a ≥ 0.5, circle N; otherwise, circle Y				
AM.3(2)	Are critical loop and multiple transfer index parameters (e.g., supporting a mission-critical function) checked for out-of-range values before use?				
AM.3(3)	Are all critical subscripts (e.g., supporting a mission-critical function) checked for out-of range values before use?				
AM.3(4)	Are all critical output data (e.g., supporting a mission-critical function) checked for reasonable values prior to final outputting?				
AM SCORE:	Count the number of Y's and N's circled and calculate the ratio of N's to the total number of N's and Y's. Assign that value to AM. TOTALS				
		AM=			

PROCEDURE NO. 3

1. Title: Traceability (ST)

2. Prediction or Estimation Parameter Supported: Software Characteristics

3. Objectives: The purpose of this metric is to determine the relationship between modules and requirements. If this relationship has been made explicit, there is greater likelihood that the modules will correctly fulfill the requirements. It should be possible to trace module characteristics to the requirements.

4. Overview: This metric indicates whether a cross reference exists which relates functions or modules to the requirements.

5. Assumptions/Constraints: The intent of the metric requires an evaluation of the correctness or completeness of the requirements matrix. It is assumed that the existence of the matrix will have a positive effect upon reliability.

6. Limitations: To achieve the true intent of this metric, a sophisticated tool or requirements specification language must be used. In its simplest form, the metric can simply be a check to see if a cross-reference matrix exists.

7. Applicability: Traceability may be determined during the requirements and design phases of the software development cycle.

8. Required inputs: Requirements and design documentation should include a cross reference matrix.

9. Required Tools: No special tools are required, however, use of a formal requirements specification language, PDL, or traceability tool provides significant savings in effort to develop this metric.

10. Data Collection Procedures: Documentation is reviewed to determine the presence or absence of the cross reference matrix, to itemize requirements at one level and their fulfillment at another. Metric Worksheet 3 can be used.

11. Outputs: Problem Reports should be written for each instance that a requirement is not fulfilled at a lower level specification.

12. Interpretation of Results: The cross reference should be taken as an indication of software quality, in that the presence of the matrix will make it more likely that implemented software actually meets requirements. Identified traceability problems should be reviewed for significance.

13. Reporting: The project engineer should be made aware of the presence or absence of the stated cross reference, to determine whether contractual requirements have been met.

14. Forms: Discrepancy Reports should be generated for all instances of lack of traceability. Metric Worksheet 3 contains checklist items for this item.

15. Instructions: The following worksheets are used to assess traceability of the software system. The worksheets should be applied as follows:

WORKSHEETS	APPLICATION
3A	During Software Requirements Analysis (at SSR)
3B	During Preliminary Design (at PDR)
3C	During Detailed Design and coding (at CDR)

Note: Complete the worksheets as follows. Calculate a value if required. Check Yes or No in response to a question. You may enter your answers directly on the worksheets or on the provided answer sheet.

Perform the following steps using **Worksheet 3A** and **Answer Sheet 2 (Appendix C)** to collect and record data for measuring **Traceability** at **SSR** for each **CSCI** of the system:

Step 2a. Review Pertinent documentation as needed (Table TS100-3).

Step 2b. Record current date for **Item II** on answer sheet, and complete items TC.1(1) and ST SCORE based upon questions in worksheet.

Perform the following steps using **Worksheet 3B** and **Answer Sheet 3 (Appendix C)** to collect and record data for measuring **Traceability** at **PDR** for each **CSCI** of the system:

Step 5a. Review pertinent documentation as needed (Table TS100-3).

Step 5b. Record current date for **Item II** on answer sheet, and complete items TC.1(1) and ST SCORE based on questions in worksheet.

Perform the following steps using **Worksheet 3C** and **Answer Sheet 5 (Appendix C)** to collect and record data for measuring **Traceability** at **CDR** for each **CSCI** of the system:

Step 10a. Review pertinent documentation as needed (Table TS100-3).

Step 10b. Record current date for **Item II** on answer sheet, and complete items TC.1(1), TC.1(2), and ST SCORE based on questions in worksheet.

16. Potential/Plans for Automation: Tools such as PSL/PSA, SREM, RTT, USE-IT assist in the determination of this metric.

METRIC WORKSHEET 3A: TRACEABILITY (1)

PHASE/REVIEW: Software Requirements Analysis/SSR
APPLICATION LEVEL: CSCI

TC.1(1)	Is there a table(s) tracing all of the CSCI's allocated requirements to the parent system or the subsystem specification(s)?
ST SCORE	If "YES," enter 1 If "NO," enter 1.1

METRIC WORKSHEET 3B: TRACEABILITY (2)

PHASE/REVIEW: Preliminary Design/PDR
APPLICATION LEVEL: CSCI

TC.1(1) Is there a table(s) tracing all the top-level CSC allocated requirements to the parent CSCI specification?

ST SCORE If "YES," enter 1
If "NO," enter 1.1

METRIC WORKSHEET 3C: TRACEABILITY (3)

PHASE/REVIEW: Detailed Design/CDR
APPLICATION LEVEL: CSCI

TC.1(1) Does the description of each software unit identify all the requirements (specified at the top-level CSC or CSCI level) that the unit helps satisfy?

TC.1(2) Is the decomposition of top-level CSCs into lower-level CSCs and software units graphically depicted?

ST SCORE If "YES" to both questions, enter 1
If "NO" to either one or both questions, enter 1.1

METRIC WORKSHEET 3D

TRACEABILITY

Itemize individual requirements and trace their flowdown through design to code. Worksheet 3D is available to trace this requirements flowdown. Contractor specified format is acceptable.

SYSTEM REQUIREMENTS	DESIGN DERIVATIVE	SOFTWARE COMPONENT
Example: SRS Para 2.4.1	SSS Para 2.4.1.1 SSS Para 2.4.1.2	PDS Para 3.10.1.1 PDS Para 3.10.1.2 PDS Para 3.10.1.3

Count Total Number of Itemized Requirements: NR = ____.

PROCEDURE NO. 4

1. Title: Quality Review (SQ)

2. Prediction or Estimation Parameter Supported: Software Characteristics

3. Objectives: This procedure consists of worksheets to assess the following characteristics:

   a. Standard design representation;

   b. Calling sequence conventions;

   c. Input/output conventions;

   d. Data naming conventions;

   e. Error handling conventions;

   f. Unambiguous references;

   g. All data references defined, computed, or obtained from all external source;

   h. All defined functions used;

   i. All conditions and processing defined for each decision point;

   j. All defined and referenced calling parameters agree;

   k. All problem reports resolved;

   l. Accuracy analysis performed and budgeted to module;

   m. A definitive statement of requirement for accuracy of inputs, outputs, processing, and constraints;

   n. Sufficiency of math library;

   o. Sufficiency of numerical methods;

   p. Execution outputs within tolerances; and

   q. Accuracy requirements budgeted to functions/modules.

   These are combined to form a metric, SQ, which represents how well these characteristics have been designed and implemented in the software system.

4. Overview: This metric will be determined at the requirements analysis and design phases of a software development. The metric itself reflects the number of problems found during reviews of the requirements and design of the system.

5. Assumptions/Constraints: Formal problem reporting during requirements and design phases of software developments has been inconsistently performed in the past. Methodologies advocated in recent years and more disciplined contractual/Government requirements and standards now encourage this activity. Assumed in this metric is a significant effort to perform formal reviews. Techniques such as Design Inspections or walk-throughs are the mechanism through which problems will be identified. Use of Worksheet 10 is also an alternative.

6. Limitations: The degree to which the requirements and design specifications are reviewed will influence the number of problems found. Consistent application of the worksheets for this procedure as a QA technique will alleviate this limitation.

7. Applicability: The primary application of this metric is to the requirements phase and design phases of the software development.

8. Required Inputs: Requirements Specification, Preliminary Design Specification, Detailed Design Specification are required.

9. Required Tools: Checklists will be used in determining this metric.

10. Data Collection Procedures: Documentation will be reviewed at the end of each phase of the system development to determine the presence or absence of these characteristics.

    Since this procedure assesses the quality at early stages of the development, it will require a comprehensive review of documentation. Detailed records must be maintained (Discrepancy Reports). Reviews will be performed using Worksheet 10.

11. Outputs: Reports of the current number of discrepancy reports (DR), together with detailed information for the project manager, will be prepared.

12. Interpretation of Results: To some extent, software will be incomplete throughout most of the development cycle, until the point at which all variables, operations, and control structures are completely defined. This metric serves, then, as a measure of progress. An incomplete software system by definition, is unfinished.

13. Reporting: Detailed reports of problems should be furnished to the project manager and the software supervisor, to assist in determining the current status of software development.

14. Forms: Worksheet 4 will be required.

15. Instructions: Worksheet 4 is used to conduct design and code reviews. These worksheets are recommended for use in conjunction with the software reliability prediction and estimation methodology. Alternative techniques that can be used are design and code inspections or design and code walk-throughs. The intent of these worksheets and these alternative techniques are to uncover discrepancies that should be corrected.

    The worksheets contained in this instruction relate to the metric worksheets in RADC TR 85-37 for metrics completeness, consistency, accuracy, autonomy, modular design and code simplicity.

    The following worksheets are used to assess the quality of the requirements and design representation of the software. Check the answer, yes, no or not applicable, or fill in the value requested in the appropriate column. The worksheets should be applied as follows:

WORKSHEET	APPLICATION
4A	During Software Requirements Analysis (at SSR)
4B	During Preliminary Design (at PDR)
4C	During Detailed Design, CSCI Level (at CDR)
4D	During Detailed Design, Unit Level (at CDR)

Note: First, complete Worksheet 4. Then complete the remaining worksheets as follows. Calculate a value if required. Check Yes or No on the line in response to a question. Check NA to a question that is not applicable and these do not count in calculation of metric. You may enter your answers directly on the worksheets or on the provided answer sheet.

Perform the following steps using **Worksheet 4A** and **Answer Sheet 2 (Appendix C)** to collect and record data for performing **Quality Review** at **SSR** for each **CSCI** of the system:

Step 3a. Review pertinent documentation as needed (Table TS100-3).

Step 3b. Record current date for **Item III** on answer sheet and complete items AC.1(3) through CS.2(6) based on questions in worksheet.

Perform the following steps using **Worksheet 4B** and **Answer Sheet 3 (Appendix C)** to collect and record data for performing **Quality Review** at **PDR** for each **CSCI** of the system:

Step 6a. Review pertinent documentation as needed (Table TS100-3).

Step 6b. Record current date for **Item III** on answer sheet and complete items AC.1(7) through CS.2(6) based on questions in worksheet.

Perform the following steps using **Worksheet 4C** and **Answer Sheet 4 (Appendix C)** to collect and record data for performing **Quality Review** at **CDR** for each **Unit** of the CSCI:

Step 8a. Review pertinent documentation as needed (Table TS100-3).

Step 8b. Record current date for **Item II** on answer sheet and complete items CP.1(1) through CS.2(6) based on questions in worksheet.

Perform the following steps using **Worksheet 4D** and **Answer Sheet 5 (Appendix C)** to collect and record data for performing **Quality Review** at **CDR** for each **CSCI** of the system:

Step 11a. Review pertinent documentation as needed (Table TS100-3).

Step 11b. Record current date for **Item III** on answer sheet, and complete items AU.1(2) through CS.2(6) based on questions in worksheet.

16. Potential/Plan for Automation: Information for this factor will be obtained manually. The IRMS may be used to automate the calculation. RADC-developed Automated Measurement System (AMS) provides checklists for use in reviewing documents.

17. Remarks: Determination of quality will require extensive review of documentation, and will thus be expensive. The extra cost may be justified if the information obtained can be used to correct faults as they are uncovered.

METRIC WORKSHEET 4
QUALITY REVIEW

GENERAL INFORMATION

1. Project ______________________________

2. Date ______________________________

3. Analyst ______________________________

4. Produc: ______________________________

5. Source Documentation

____

____

____

____

METRIC WORKSHEET 4A: QUALITY REVIEW (1)

PHASE/REVIEW: Software Requirements Analysis/SSR
APPLICATION LEVEL: CSCI

		Yes	No	NA	Unk
AC.1(3)	Are there quantitative accuracy requirements for all applicable inputs associated with each applicable function (e.g., mission-critical functions)?				
AC.1(4)	Are there quantitative accuracy requirements for all applicable outputs associated with each applicable function (e.g., mission-critical functions)?				
AC.1(5)	Are there quantitative accuracy requirements for all applicable constants associated with each applicable function (e.g., mission-critical functions)?				
AC.1(6)	Do the existing math library routines which are planned for use provide enough precision to support accuracy objectives?				
AU.1(1)	Are all processes and functions partitioned to be logically complete and self contained so as to minimize interface complexity?				
AU.2(1)	Are there requirements for each operational CPU/System to have a separate power source?				
AU.2(2)	Are there requirements for the executive software to perform testing of its own operation and of the communication links, memory devices, and peripheral devices?				
CP.1(1)	Are all inputs, processing, and outputs clearly and precisely defined?				
CP.1(2)	a. How many data references are identified?				
	b. How many identified data references are documented with regard to source, meaning, and format?				
	c. Calculate b/a.				
	d. If b/a < 1, circle N If b/a = 1, circle Y				
CP.1(3)	a. How many data items are defined (i.e., documented with regard to source, meaning, and format)?				
	b. How many data items are referenced?				
	c. Calculate b/a.				
	d. If b/a < 1, circle N If b/a = 1, circle Y				

METRIC WORKSHEET 4A: QUALITY REVIEW (1) (cont.)

PHASE/REVIEW: Software Requirements Analysis/SSR
APPLICATION LEVEL: CSCI

		Yes	No	NA	Unk
CP.1(5)	Have all defined functions (i.e., documented with regard to source, meaning, and format) been referenced?				
CP.1(6)	Have all system functions allocated to this CSCI been allocated to software functions within this CSCI?				
CP.1(7)	Have all referenced functions been defined (i.e., documented with precise inputs, processing, and output requirements)?				
CP.1(8)	Is the flow of processing (algorithms) and all decision points (conditions and alternate paths) in the flow described for all functions?				
CS.1(1)	Have specific standards been established for design representations (e.g., HIPO charts, program design language, flow charts, data flow diagrams)?				
CS.1(2)	Have specific standards been established for calling sequence protocol between software units?				
CS.1(3)	Have specific standards been established for external I/O protocol and format for all software units?				
CS.1(4)	Have specific standards been established for error handling for all software units?				
CS.1(5)	Do all references to the same CSCI function use a single, unique name?				
CS.2(1)	Have specific standards been established for all data representation in the design?				
CS.2(2)	Have specific standards been established for the naming of all data?				
CS.2(3)	Have specific standards been established for the definition and use of global variables?				
CS.2(4)	Are there procedures for establishing consistency and concurrency of multiple copies (e.g., copies at different nodes) of the same software or data base version?				
CS.2(5)	Are there procedures for verifying consistency and concurrency of multiple copies (e.g., copies at different nodes) of the same software or data base version?				
CS.2(6)	Do all references to the same data use a single, unique name? TOTALS				
SQ INPUT: Count all N's. Assign number to DR.		DR=			

METRIC WORKSHEET 4B: QUALITY REVIEW (2)

PHASE/REVIEW: Preliminary Design/PDR
APPLICATION LEVEL: CSCI

		Yes	No	NA	Unk
AC.1(7)	Do the numerical techniques used in implementing applicable functions (e.g., mission-critical functions) provide enough precision to support accuracy objectives?				
AU.1(1)	Are all processes and functions partitioned to be logically complete and self-contained so as to minimize interface complexity?				
AU.1(4)	a. How much estimated process time is typically spent executing the entire CSCI?				
	b. How much estimated processing time is typically spent in execution of hardware and device interface protocol?				
	c. Calculate b/(b+a).				
	d. If b/(b + a) > .3, circle N. If b/(b + a) < .3, circle Y.				
AU.2(2)	Does the executive software perform testing of its own operation and of the communication links, memory devices, and peripheral devices?				
CP.1(1)	Are all inputs, processing, and outputs clearly and precisely defined?				
CP.1(2)	a. How many data references are defined?				
	b. How many identified data references are documented with regard to source, meaning, and format?				
	c. Calculate b/a.				
	d. If b/a ≤ 0.5, circle N. If b/a < 0.5, circle Y.				
CP.1(3)	a. How many data items are defined (i.e., documented with regard to source, meaning, and format)?				
	b. How many data items are referenced?				
	c. Calculate b/a.				
	d. If b/a ≤ 0.5, circle N. If b/a > 0.5, circle Y.				

METRIC WORKSHEET 4B: QUALITY REVIEW (2) (cont.)

PHASE/REVIEW: Preliminary Design/PDR
APPLICATION LEVEL: CSCI

		Yes	No	NA	Unk
CP.1(4)	a. How many data references are identified? b. How many identified data references are computed or obtained from an external source (e.g., referencing global data with preassigned values, input parameters with preassigned values)? c. Calculate b/a. d. If b/a ≤ 0.5, circle N				
CP.1(6)	Have all functions for this CSCI been allocated to top-level CSCs of this CSCI? If b/a > 0.5, circle Y				
CP.1(9)	Are all conditions and alternative processing options defined for each decision point?				
CP.1(11)	a. How many software discrepancy reports have been recorded, to date? b. How many recorded software problem reports have been closed (resolved), to date? c. Calculate b/a. d. If b/a 0.75, circle N If b/a > 0.75 circle Y				
CS.1(1)	Are the design representations in the formats of the established standard?				
CS.1(5)	Do all references to the same top-level CSC use a single, unique name?				
CS.2(1)	Does all data representation comply with the established standard?				
CS.2(2)	Does the naming of all data comply with the established standard?				
CS.2(3)	Is the definition and use of all global variables in accordance with the established standard?				
CS.2(4)	Are there procedures for establishing consistency and concurrency of multiple copies (e.g., copies at different nodes) of the same software or data base version?				
CS.2(5)	Are there procedures for verifying consistency and concurrency of multiple copies (e.g., copies at different nodes) of the same software or data base version?				
CS.2(6)	Do all references to the same data use a single, unique name?				
SQ INPUT: Count all N's. Assign to DR.					
	TOTALS				
		DR=			

METRIC WORKSHEET 4C: QUALITY REVIEW (3)

PHASE/REVIEW: Detailed Design/CDR
APPLICATION LEVEL: Unit

		Yes	No	NA	Unk
CP.1(1)	Are all inputs, processing, and outputs clearly and precisely defined?				
CP.1(2)	a. Are all data references are defined?				
	b. How many identified data references are documented with regard to source, meaning, and format?				
CP.1(4)	a. Are all data references identified? (see CP.1(2)a above)				
CP.1(9)	Are all conditions and alternative processing options defined for each decision point?				
CP.1(10)	Are all parameters in the argument list used?				
CS.1(1)	Are all design representations in the formats of the established standard?				
CS.1(2)	Does the calling sequence protocol (between units) comply with established standard?				
CS.1(3)	Does the I/O protocol and format comply with the established standard?				
CS.1(4)	Does the handling of errors comply with the established standard?				
CS.1(5)	Do all references to this unit use the same, unique name?				
CS.2(1)	Does all data representation comply with the established standard?				
CS.2(2)	Does the naming of all data comply with the established standard?				
CS.2(3)	Is the definition and use of all global variables in accordance with the established standard?				
CS.2(6)	Do all references to the same data use a single, unique name?				
SQ INPUT: Count all No answers. Assign to DR.	TOTALS				
		DR=			

METRIC WORKSHEET 4D: QUALITY REVIEW (4)

PHASE/REVIEW: Detailed Design/CDR
APPLICATION LEVEL: CSCI

		Yes	No	NA	Unk
AU.1(2)	a. How many estimated executable lines of source code? (total from all units)				
	b. How many estimated executable lines of source code necessary to handle hardware and device interface protocol?				
	c. Calculate b/a.				
	d. If b/a > .3, circle N. If b/a ≤ .3, circle Y.				
AU.1(3)	a. How many units (NM) in CSCI?				
	b. How many units perform processing of hardware and/or device interface protocol?				
	c. Calculate b/NM.				
	d. If b/NM > .3, circle N. If b/NM ≤ .3, circle Y.				
AU.1(4)	a. How much estimated processing time is typically spent executing the entire CSCI?				
	b. How much estimated processing time is typically spent in execution of hardware and device interface protocol units?				
	c. Calculate b/a.				
	d. If b/a > .3, circle N. If b/a ≤ .3, circle Y.				
CP.1(1)	a. How many units clearly and precisely define all inputs, processing, and outputs?				
	b. Calculate a/NM.				
	c. If a/NM ≤ 0.5, circle N. If a/NM > 0.5, circle Y.				

METRIC WORKSHEET 4D: QUALITY REVIEW (4) (cont.)

PHASE/REVIEW: Detailed Design/CDR
APPLICATION LEVEL: CSCI

		Yes	No	NA	Unk
CP.1(2)	a. How many data references are identified? (total from all units)				
	b. How many identified data references are documented with regard to source, meaning, and format? (total from all units)				
	c. Calculate b/a.				
	d. If b/a ≤ 0.5, circle N. If b/a > 0.5, circle Y.				
CP.1(3)	a. How many data items are defined (i.e., documented with regard to source, meaning, and format)?				
	b. How many data items are referenced?				
	c. Calculate b/a.				
	d. If b/a ≤ 0.5, circle N. If b/a > 0.5, circle Y.				
CP.1(4)	a. How many data references are identified? (from CP1(2)a above)				
	b. How many identified data references are computed or obtained from an external source (e.g., referencing global data with preassigned values, input parameters with preassigned values)? (total from all units)				
	c. Calculate b/a.				
	d. If b/a ≤ 0.5, circle N. If b/a > 0.5, circle Y.				
CP.1(9)	a. How many units define all conditions and alternative processing options for each decision point? (total from all units)				
	b. Calculate a/NM.				
	c. If a/NM ≤ 0.5, circle N. If a/NM > 0.5, circle Y.				
CP.1(10)	a. For how many units, are all parameters in the argument list used?				
	b. Calculate a/NM.				
	c. If a/NM ≤ 0.5, circle N. If a/NM > 0.5, circle Y.				

METRIC WORKSHEET 4D: QUALITY REVIEW (4) (cont.)

PHASE/REVIEW: Detailed Design/CDR
APPLICATION LEVEL: CSCI

		Yes	No	NA	Unk
CP.1(11)	a. How many software problem reports have been recorded, to date?				
	b. How many recorded software problem reports have been closed (resolved), to date?				
	c. Calculate b/a.				
	d. If c 0.75, circle N. If c > 0.75, circle Y.				
CS.1(1)	a. For how many units are all design representations in the formats of the established standard?				
	b. Calculate a/NM.				
	c. If a/NM ≤ 0.5 circle N, otherwise circle Y.				
CS.1(2)	a. For how many units does the calling sequence protocol (between units) comply with the established standard?				
	b. Calculate a/NM.				
	c. If a/NM ≤ 0.5 circle N, otherwise circle Y.				
CS.1(3)	a. For how many units does the I/O protocol and format comply with the established standard?				
	b. Calculate a/NM.				
	c. If a/NM ≤ 0.5 circle N, otherwise circle Y.				
CS.1(4)	a. For how many units does the handling of errors comply with the established standard?				
	b. Calculate a/NM.				
	c. If a/NM ≤ 0.5 circle N, otherwise circle Y.				
CS.1(5)	a. For how many units do all references to the unit use the same, unique name?				
	b. Calculate a/NM.				
	c. If a/NM < 1, circle N, otherwise circle Y.				

METRIC WORKSHEET 4D: QUALITY REVIEW (4) (cont.)

PHASE/REVIEW: Detailed Design/CDR
APPLICATION LEVEL: CSCI

		Yes	No	NA	Unk
CS.2(1)	a. For how many units does the naming of ali data comply with the established standard?				
	b. Calculate a/NM.				
	c. If a/NM ≤ 0.5 circle N, otherwise circle Y.				
CS.2(2)	a. For how many units does the naming of all data comply with the astablished standard?				
	b. Calculate a/NM.				
	c. If a/NM ≤ 0.5 circle N, otherwise circle Y.				
CS.2(3)	a. For how many units is the definition and use of all global variables in accordance with the established standard?				
	b. Calculate a/NM.				
	c. If a/NM ≤ 0.5 circle N, otherwise circle Y.				
CS.2(6)	a. For how many units do all references to the same data use a single, unique name?				
	b. Calculate a/NM.				
	c. If a/NM < 1 circle N, otherwise circle Y.				
	SQ SCORE: Total the number of No's. Assign to DR.				
	TOTALS				
		DR=			

PROCEDURE 5

1. Title: Size Estimation (NR and SLOC).

2. Prediction or Estimation Parameter Supported: Fault Density.

3. Objectives: To determine fault density, some measure of size must be used as the sample size (denominator). Described here are two alternatives:

   a. Number of system requirements (this is appropriate for Task 103) and
   b. Number of source lines of code (during Task 104).

4. Overview: During the early phases of a development, problems identified are typically at a system or subsystem level. In order to provide some relative measure of the significance of these problems, a sizing measure is needed at a system level. A simple measure of size is the number of functions required in the System Specification (the number of shall statements may accurately reflect this).

   Later in the development, an estimate or actual count (during coding) of the number of lines of code will provide a basis for judging problems identified at the module level. Program size is not generated automatically by operating system software, since the number of printed lines may include comments, declarations, blank lines, or lines containing multiple statements. Our accepted definition of source lines of code will be the number of executable statements.

5. Assumptions/Constraints: It is assumed that the number of executable statements can be compared among systems. This assumption is not likely to hold when systems are written in different languages: a comparison of FORTRAN with LISP or APL would be misleading because of the greater compactness of LISP and APL in many applications. However, most of the HOLs to be considered are similar enough to make this metric sufficiently reliable for estimates of program size.

6. Limitations: Counting the number of requirements involves significant discipline. Use of a formal requirements specification language simplifies the task significantly. Use of the concept of function points is another alternative. The key is to be consistent.

7. Applicability: Estimates of program size should be available during all development phases.

8. Required Inputs: The Size Estimates will be based on the Requirements Specifications and the software.

9. Required Tools: Requirements Specification languages or analysis tools such as PSL/PSA, SREM, RTT, USE-IT are applicable. Compilers or code audit routines generally provide lines of code counts.

10. Data Collection Procedures: To determine the number of requirements, individual requirements must be itemized by analysis of the Requirements Specification. Data Collection Worksheet 3D can be used.

    Refer to Procedure No. 3 to complete Worksheet 3D (page B26) for reporting Number of Requirements (NR) during Task 103.

To estimate lines of code, use of senior personnel familiar with the specific application or reference to a historical data base which provides code counts for certain applications are the most proven techniques. Metric Collection Worksheet 7A on page B-80 can be used.

Use of the compiler output or code auditors provide actual counts once coding is underway.

11. Outputs: Program size (SLOC) and number of requirements (NR) are reported.

12. Interpretation of Results: Program size can be used as a predictor of error rates. However, its primary use in this research is in combination with Software Discrepancy Reports in determining fault density

13. Reporting: Program size is reported to the project manager as required for estimating resource requirements.

14. Forms: Worksheet 3D on page B-26 or Metric Worksheet 7A (page B-80) provides for reporting Program Size during Task 104.

15. Potential/Plans for Automation: Moderate revisions of existing system software should make it possible to obtain more accurate counts of program size in terms of number of lines of executable code.

16. Remarks: As noted, the measurement of program size has been used in the past as a predictor of software quality. Program size should be correlated with software failure rates, where appropriate, to determine the significance of this metric.

PROCEDURE NO. 6

1. Title: Fault Density

2. Prediction or Estimation Parameter Supported: Fault Density

3. Objectives: Fault Density represents a measure of the number of faults in a software system.

4. Overview: Fault Density may be used to provide a preliminary indication of software reliability. Because of the functional relationship between this metric and the Failure Rate, it provides an alternative measure of software reliability. Its major advantages are that it is fairly invariant and that it can be obtained from commonly available data.

5. Assumptions/Constraints: The predicted fault density will depend in part on the review and test procedures used to detect software faults. In any case, there is no guarantee that all faults have been found. Although it can be used to estimate failure rates, it cannot be directly combined with hardware reliability metrics.

6. Limitations: As noted, the Fault Density estimates may be affected by the review and testing procedures.

7. Applicability: This number is confinned during the formal testing phases where faults are observed and discrepancy reports formally recorded. During early phases of the development, a fault density measure can be obtained by using the number of problem reports documented during reviews or the prediction methodology.

8. Required Inputs: Estimates of Fault Density are obtained from software discrepancy reports. The number of faults reported, divided by the number of lines of executable code (or number of requirements during early phases of development), gives the required metric. Reference is made to Data Collection procedures 6 and 7.

9. Required Tools: Accurate records of software faults are essential for this metric. A data base management system to prepare summary reports would simplify record keeping and preparation of calculation of Fault Density.

10. Data Collection Procedures: A count of software faults is obtained through inspection of software discrepancy reports. The number of lines of executable code will also be required. Use of a discrepancy report such as that at Worksheet 6 is recommended.

11. Outputs: The predicted Fault Density (RP) is the primary output. In addition, estimates of failure rates, based on the transformations described in Task 100, will also be output.

12. Interpretation of Results: The Fault Density is used as a predictor for the Failure Rate, and thus should provide an important indicator of software reliability in advance of full-scale system tests. It also can be compared with a specified fault density as a requirement or with industry averages represented in Table TS101-1. It is also an indicator of individual components that are potentially high risk elements or unreliable components.

13. Reporting: This metric is reported, together with the estimates of failure rates, to support predictions and estimates of software reliability.

14. Forms: Metric Worksheet 6.

15. Instructions: During the Quality Review, Standards Review, or equivalent reviews such as Design and Code Inspections or Walk-throughs; during formal reviews such as SRR, PDR, CDR; and during testing, problems should be formally documented. The Discrepancy Report (Worksheet 6) or an equivalent problem report form should be used. The discrepancy report records the following information:

    a. Problem title and ID
    b. Analyst who uncovered problem
    c. Date it was found and phase of development
    d. Type of Problem
    e. Criticality of Problem
    f. How it was detected
    g. Description of Problem
    h. What test run and how much test time was expended if it was found during testing
    i. Impact of Problem
    j. Solution
    k. Acknowledgement that it is a problem and date
    l. Acknowledgement that it has been fixed and dated

16. Potential/Plans for Automation: The software discrepancy reports may be kept in standard formats for access through a data base management system. The system should be sufficiently powerful to provide counts of errors for each module and to calculate fault densities, if the module lengths are available.

17. Remarks: The fault density, because of its functional relationship to failure rates, will provide an estimate of software reliability during coding and early testing.

WORKSHEET 6 DISCREPANCY REPORT

PROBLEM TITLE:

PROBLEM NUMBER:

DATE:

PROGRAM ID:

ANALYST:

REFERENCES:

PROBLEM TYPE:

REQUIREMENTS

- Incorrect Spec
- Conflicting Spec
- Incomplete Spec

DESIGN

- Requirements Compliance
- Choice of Algorithm
- Sequence of Operations
- Data Definitions
- Interface

CODING

- Requirements or Design Compliance
- Computation Implementation
- Sequence of Operation
- Data Definition
- Data Handling
- Omitted Logic
- Interface
- Performance

MAINTENANCE

- Incorrect Fix
- Incompatible Fix

OTHER

CRITICALITY

HIGH MEDIUM LOW

METHOD DETECTION:

DESCRIPTION OF PROBLEM:

TEST EXECUTION: **TEST CASE ID:** **TEST EXECUTION TIME:**

EFFECTS OF PROBLEM:

RECOMMENDED SOLUTION:

APPROVED:

RELEASED BY:

DATE:

DATE:

PROCEDURE NO. 7

1. Title: Discrepancy Reports (DR)

2. Prediction or Estimation Parameter Supported: Fault Density and Failure Rate

3. Objectives: The basic metric for estimation will be the observed failure rate during testing During Operation and Maintenance, the observed failure rate will also be used. The failure rate is based on the observed number of failures over time, which is derived from Discrepancy Reports and Execution Time measures.

4. Overview: A Software Discrepancy Report is generated at the time that an error is discovered or a failure occurs, typically during formal testing. An error is a discrepancy between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition. A failure occurs when the system or system component is unable to perform a required function within specified limits. A count of failures will be obtained from the Discrepancy Reports.

5. Assumptions/Constraints: Reported failure rates will not accurately reflect actual failure frequencies unless procedures for preparing and recording software problems are strictly enforced by project management. It is necessary to assume that differences in reported failure rates reflect actual differences between software components. Care must be taken to ensure that these differences are not merely artifacts of the collection procedures.

6. Limitations: Software induced failures will differ in seriousness, ranging from low-priority (easily corrected or avoided) to high-priority (results in mission failure). This information should appear on Discrepancy Reports, although it is not presently used directly in determining failure rates. The recommended categories of High, Medium and Low are defined in paragraph 8 below. Further research in the utilization of severity as a prediction criterion is warranted.

7. Applicability: Discrepancy Reports will be obtained during any formal reviews, coding and unit testing, CSC integration and testing, CSCI-level testing, acceptance testing, operational test and evaluation and O&M. MIL-STD 2167A references Software Problem Reports as backup to the Software Test Result Report. The Discrepancy Report described here meets that requirement as well as provides a mechanism for recording other discrepancies identified formally.

8. Required Inputs: Discrepancy Reports are documented by the program development staff, QA, customer testers or by the O&M staff as problems occur or are identified. Specific procedures are to be included in the Software Quality Assurance Plan and Configuration Management Plan for the system. The Discrepancy Report contains an identification section in which a title and identification number are entered as well as as the author, date, and any references that should be included. It also is recommended that a discrepancy report includes a categorization scheme that will support trend analyses. The discrepancy report recommended in this guidebook (see Metric Worksheet 6) categorizes the discrepancy by type and criticality level. The criticality levels are:

High	causes system to abort or fail to perform mission.
Medium	incorrect results are obtained but does not necessarily jeopardize mission.

Low typically involves incorrect format, documentation errors, or miscalculations that does not threaten mission performance but should be fixed eventually.

Also described by the discrepancy report are the method used to detect the discrepancy, a description of the problem, the impact or effects of the problem, the recommended solution, and data on the test case and execution time, if the discrepancy was found during a test run. Discrepancy reports usually are approved after the appropriate fix has been made and QA releases it to configuration management for formal update of the current version of the software.

9. Required Tools: On-line entry of Discrepancy Reports will require storage of appropriate formats for the reports, and subsequent storage and retrieval facilities. Automatic computation of failure rates for system components is desirable.

10. Data Collection Procedures: Discrepancy Reports will be collected during system tests and operation as one of the responsibilities of the project manager.

11. Outputs: Discrepancy Reports will be accessible in a designated file. Their primary relevance will be in determination of the Failure Rate.

12. Interpretation of Results: Discrepancy Reports play a central role in the validation of software reliability metrics. The Failure Rate, based on information obtained from the reports, provides the baseline against which metrics for prediction and estimation are validated.

13. Reporting: For the purposes of this research, the Failure Rate will be reported. Since the Discrepancy Reports contain additional information of interest to project managers, they will be available for further reference and analysis.

14. Forms: A standard Discrepancy Report form is recommended (see Data Collection Worksheet 6).

15. Potential/Plans for Automation: Discrepancy Reports are stored and retrieved through the file management system, but are prepared manually.

16. Remarks: Because of the importance of accurate and complete Discrepancy Reports in determination of failure rate, the collection and maintenance of the reports should be included in the management plan for any software included in the current reliability research project. Comparisons of failure rates between two systems will be misleading unless the same criteria have been used for both systems.

PROCEDURE NO. 8

1. Title: Language Type (SL)

2. Prediction or Estimation Parameter Supported: Software Characteristics

3. Objectives: Categorizes language or languages used in software unit as assembly or higher order language (HOL).

4. Overview: In the Language Type metric, the system is categorized according to language. Language Type has been shown to have an effect on error rates.

5. Assumptions/Constraints: Because of the significant effect that language can have on software reliability, use of this metric will provide an early indication of expected failure rates.

   Dunng the requirements phase, language requirements may be tentatively indicated, particularly when a new system must interface with existing software or hardware.

   During the specifications phase, detailed information concerning proportions of HOL and assembly code will normally become available.

   Finally, during integration and test, it may become necessary to change the specified proportion of assembly code in order to meet space, time, or performance constraints.

6. Limitations: Accuracy of this metric will depend on the accuracy of estimates of lines of HOL and assembly language code during early phases of development. While detailed specifications will normally include an estimate of program size, this estimate must be revised during software development.

7. Applicability: This metric is obtained during the preliminary design phase to provide an early warning of potential effects of language selection. Because of the higher error rates encountered when assembly language programming is used, it may indicate a choice of HOL rather than assembly language.

   More importantly, it can provide a measure of the cost, in terms of higher error rates, to be balanced against projected savings in time and space, for a proposed design.

8. Required Inputs: Information is extracted manually from requirements or specifications documentation. During implementation and test, more accurate measures of the number of lines of code will be available from compilers or automated program monitors.

9. Required Tools: Information is extracted manually from existing documentation during requirements and specifications phases. During implementation and test, information will be available from compiler output or code auditors.

10. Data Collection Procedures: Initial estimates of lines of code will be extracted from existing documentation. When this information is not available, the value of the metric will be set to 1.0. Counts of the number of lines of source code may be obtained from compilations of software units. Comments and blank lines should not be included in this total, and it may be necessary to exclude them manually.

11. Outputs: The following outputs are required from this procedure,

ALOC = The number of lines in assembly language

HLOC = The number of lines in HOL

SLOC = ALOC + HLOC = total number of executable lines of code (see Data Collection Procedure 6).

These are combined according to the following formula:

$$SL = HLOC/SLOC + 1.4 * ALCO/SLOC$$

12. Interpretation of Results: When combined with other metrics, SL will indicate the degree to which the predicted or estimated error rate will be increased because of the use of assembly language. This information, when compared with the expected increase in efficiency through the use of assembly language, can be used as a basis for a decision concerning implementation language.

13. Reporting: The value of SL will be reported and combined with other measures in obtaining a predicted failure rate.

14. Forms: Worksheet 8D for reporting the number of lines of code, the proportion lines in each stated category, and the composite SL.

15. Instructions: Answer questions in Metric Worksheet 8D and Answer sheet 6 (Appendix C).

16. Potential/Plans for Automation: Language Type will normally be specified in requirements and specifications, and must be obtained manually.

17. Remarks: As research progresses, it may become possible to make finer distinctions among languages, and among versions of the same language. For this reason, the specific implementation should be included in this report. That is, the name of the language, the version, the operating system and version, and the processor name and type should be reported when this information is available.

METRIC WORKSHEET 8D: Language Type

PHASE: Coding and Unit Testing
APPLICATION LEVEL: Unit

1a. How many modules are there in this CSCI (NM)?

b. How many executable lines of code (SLOC) are present in each unit? (AMS AU.1(2e)).

c. How many assembly language lines of code (ALOC) are present in each unit? (AMS AP.3(4e)).

d. Calculate a - b for HLOC (higher order language lines of code) for each unit.

2. Determine complexity (sx) for this unit by adding 1 to the value from AMS SI.4(11e), which then provides the following:

sx = # conditional branching stmts + # unconditional branching stmts +1

PROCEDURE NO. 9

1. Title: Module Size (SM)

2. Prediction or Estimation Parameter Supported: Software Characteristics

3. Objectives: Structured programming studies and Government procurement documents have frequently prescribed limits on module size, on the basis of the belief that smaller modules were more easily understood, and would therefore, be less likely to contain logical errors. This metric provides an estimate of the effect of module size, based on the proportions of modules with number of lines of executable code as follows:

No. of Modules	
u	Less than 100
w	100 to 500
x	Over 500

4. Overview: Inspection of compiler reports, editors, or source code will provide module length. Lines of code are counted on the same basis as that used in the Program Size metric.

5. Assumptions/Constraints: Lines of code include executable instructions. Comments and blank lines are excluded. Declarations, data statements, common statements, and other non-executable statements are not included in the total line count. Where single statements extend over more than one printed line, only one line is counted. If more than one statement is included on a printed line, the number of statements is counted.

   Assembly language lines are converted to HOL line equivalents by dividing by an appropriate expansion factor, and program size is reported in source code lines or equivalents.

6. Limitations: The precision of the reported Module Size may be affected by human factors, if the reporter is required to count lines visually, or to revise the figure reported by the compiler or editor. When the project is large enough to support it, an automatic line counter, which would produce consistent line counts, should be supplied.

7. Applicability: This metric will not be available until detailed program specifications have been written. Estimates of module size will normally be included in specifications.

8. Required Inputs: Specifications containing module size estimates may be used for early computation of this metric. As modules are completed, more accurate figures for size will become available. For existing software, module size is normally contained in system documentation; otherwise, it may be obtained through inspection of the code.

9. Required Tools: The compiler or editor will provide counts of the total number of lines in each module. Additional software tools could be provided to count lines of executable code, excluding comments and blank lines.

10. Data Collection Procedures: Compiler or editor output is examined to determine sizes for each module. Where counts include comments or blank lines, these must be eliminated to obtain a net line count. Modules are then categorized as shown above, and a count is made of the number of modules in each category.

11. Outputs: Results are reported in terms of the raw counts of the number of modules in each category, together with the resulting metric SM.

12. Interpretation of Results: In general, it has been assumed that any large modules will increase the potential failure rate of a software system. Later experiments will test this assumption.

13. Reporting: The values of u, w, and x will be reported.

14. Forms: Metric Worksheet 9D and Answer Sheet 7 (Appendix C).

15. Potential/Plans for Automation: Compilers and editors typically provide enough data to compute this metric. A fully automated system would give more accurate estimates of the number of executable statements.

16. Remarks: More sophisticated measures of modularity should be explored.

METRIC WORKSHEET 9D: LANGUAGE TYPE/COMPLEXITY/MODULARITY

PHASE: Coding and Unit Testing
APPLICATION LEVEL: CSCI

LANGUAGE TYPE (SL)

1 a. How many executable lines of code (LOC) ar present in this SCSI? (Total "1a" values for MLOC from Worksheet 4A for all units).

b. How many assembly language lines of code (ALOC) are present in this CSCI? (Total "1b" values for ALOC from Worksheet 4A for all units).

c. How many higher order language lines of code (HLOC) are present in this CSCI? (Total "1c" values for HLOC from Worksheet 4A for all units).

COMPLEXITY (SX)

2 a. For how many units in CSCI is $sx > 20$? (Refer to "2" in Worksheet 4A).

b. For how many units in CSCI is $7 \leq sx \leq 20$? (refer to "2" in Worksheet 4A).

c. For how many units in CSCI is $sx < 7$? (Refer to "2" in Worksheet 4A).

d. How many total units (NM) are present in CSCI?

MODULARITY (SM)

3 u. For how many units in system is $SLOC \leq 200$? (Refer to "1a" of Worksheet 4A).

w. For how many units in system is $200 \leq SLOC \geq 3000$? (Refer to "1a" of Worksheet 4A).

x. For how many units in system is $SLOC > 3000$? (Refer to "1a" of Worksheet 4A).

PROCEDURE NO. 10

1. Title: Complexity (SX)

2. Prediction or Estimation Parameter Supported: Software Characteristics

3. Objectives: The logical complexity of a software component relates the degree of difficulty that a human reader will have in comprehending the flow of control in the component. Complexity will, therefore, have an effect on software reliability by increasing the probability of human error at every phase of the software cycle, from initial requirements specification to maintenance of the completed system. This metric provides an objectively defined measure of the complexity of the software component for use in predicting and estimating reliability.

4. Overview: The metric may be obtained automatically from AMS, where complexity = number of branches in each module.

5. Assumptions/Constraints: Some analogue of the complexity measure might be obtained during early phases -- for example, through a count of the number of appearances of THEN and ELSE in a structured specification or by counting branches in a Program Design Language description of the design - but actual complexity can be measured only as code is produced at software implementation.

6. Limitations: Another limitation may be found in the possible interaction of this metric with length - longer programs are likely to be more complex than shorter programs -- with the result that this metric simply duplicates measurements of length.

7. Applicability: Complexity measures are widely applicable across the entire software development cycle. Reliability metrics have not yet been defined for the Requirements phase, and probably cannot be applied unless a formalized requirements language is used. To the extent that specifications have been formalized, a complexity metric may be used. The metric to be used here may be extracted automatically from code as it is produced. A series of measures will be taken over time, and increases or decreases in complexity will be noted.

8. Required Inputs: Coded modules are input to a program for complexity measurement.

9. Required Tools: An analysis program, such as AMS, capable of recognizing and counting program branches (IF-THEN-ELSE, GOTO, etc.).

10. Data Collection Procedures: If an automated tool is used, it should be possible to initiate collection simply by providing the filename of the code to be analyzed. A manual approach would use a visual count of the number of edges or paths in a flowchart representation of the modules. Another approach would be to count the number of appearances of THEN, ELSE, GOTO, WHILE, and UNTIL together with a count of the number of branches following a CASE, computed GOTO, or Fortran IF statements.

11. Outputs: An complexity measure (SX) will be output.

12. Interpretation of Results: A large value for SX indicates a complex logical structure, which will affect the difficulty that a programmer will have in understanding the structure. This in turn will affect the reliability and maintainability of the software, since the probability of human error will be higher.

13. Reporting: Abnormally high values for SX should be reported to the program managers as an indication that the system is overly complex, and thus difficult to comprehend and error prone. Complex individual modules will also be identified by high values for sx(i).

14. Forms: The report form for each module and for the system as a whole should indicate the complexity, obtained either from an automated procedure or by hand. Metric Worksheet 9D provides for reporting Complexity.

15. Instruction: Several of the measures used in the prediction methodology require sizing data about the software at various levels of detail. Such information as the overall size of the system and how it is decomposed into CSCI's, CSC's, and units, is required. Initially during a development, these data are estimates, then as the code is implemented, the actual size can be determined. Worksheet 10D can be used to tally unit data required by Data Collection Procedures 6, 8, 9 and 10. An answer sheet should be filled out for each unit and CSCI. Each unit's (MLOC) size and complexity (sx(i)) is recorded. An indication of the number of lines of higher order language (H) and assembly language (A) for each unit should be provided. The size data should be summed for all units in a CSC and for all CSC's in a CSCI.

    Complexity (SX(i)) is calculated as follows:

    (1) Count the number of conditional branch statements in a unit (eg., If, While, Repeat, DO/FOR LOOP, CASE).

    (2) Count the number of unconditional branch statements in a unit (eg., GO TO, CALL, RETURN).

    (3) Add (1) and (2)

16. Potential/Plans for Automation: A code auditor should be obtained or written to provide automated estimates of program complexity.

17. Remarks: Further experimentation with complexity metrics is desirable, and any automated tools written for this purpose should include alternative approaches, such as Halstead's metrics.

WORKSHEET 10D
SIZE/COMPLEXITY/LANGUAGE DATA

CSCI NAME =	DATE:			
CSC NAME	SLOC	UNIT NAME	MLOC	COMPLEXITY (sx(i))
CSCI SLOC=		TOTAL NO. OF UNITS: NM=		Σsx(i)=
TOTAL NO. OF HOL LOC: HLOC=		NO. OF UNITS < 100 LOC: U=		NO OF UNITS
TOTAL NO. OF AL LOC: ALOC=		NO. OF UNITS BETWEEN 100 AND 500 LOC: W=		sx(i) ≥ 20: a= 20 > Sx(i) ≥ 7: b= sx(i) < 7: c=
		NO. OF UNITS > 500 LOC: X=		

PROCEDURE NO. 11

1. Title: Standards Review (SR)

2. Prediction or Estimation Parameter Supported: Software Characteristics

3. Objectives: This metric represents standards compliance by the implementers. The code is reviewed for the following characteristics:

   a. Design organized in top-down fashion,

   b. Independence of module,

   c. Module processing not dependent on prior processing,

   d. Each module description includes input, output, processing, limitations,

   e. Each module has a single entry and at most one routine and one exception exit.

   f. Size of data base,

   g. Compartmentalization of data base,

   h. No duplicate functions, and

   i. Minimum use of global data.

4. Overview: The purpose of this procedure is to obtain a score indicating the conformance of the software with good software engineering standards and practices.

5. Assumptions/Constraints: This data will be collected via QA reviews/walk-throughs of the code or audits of the Unit Development Folders or via a code auditor developed specifically to audit the code for standards enforcement.

6. Limitations: In general, components of this metric must be obtained manually and are thus subject to human error. However, the measures have been objectively defined and should produce reliable results. The cost of obtaining these measures, where they are not currently available automatically, may be high.

7. Applicability: This data is collected during the detailed design and more readily during the coding phase of a software development.

8. Required Inputs: Code

9. Required Tools: A code auditor can help in obtaining some of the data elements.

10. Data Collection Procedures: Use Metric Worksheet 11D and review (walk-through) code.

11. Outputs: The number of modules problems with (PR) is identified.

12. Interpretation of Results: Noncompliance with standards not only means the code is probably complex, but it is symptomatic of an undisciplined development effort which will result in lower reliability.

13. Reporting: The modules which do not meet standards are reported via problem reports.

14. Forms: Metric worksheet 11D may be used and Answer Sheet 8 (Appendix C).

15. Instructions: First, complete Worksheet 11. Then complete worksheet 11D as follows. Enter a value if required on the line next to the question in the Yes column. Check Yes or No on the line if question requires a yes or no response. Check NA to a question that is not applicable and these do not count in calculation of metric. The first portion of 11D is applied to units. The second portion utilizes the units results to accumulate the answers at CSCI level.

16. Potential/Plans for Automation: In general, components of this procedure are inappropriate for manual collection. Implementation data can be collected automatically. The RADC Automated Measurement System (AMS) may be used to collect each of the data items shown in Table TS100-8.

17. Remarks: Modification of the Metric Worksheet 11 may be necessary to reflect different standards due to environment, application, or language.

METRIC WORKSHEET 11
STANDARDS REVIEW

GENERAL INFORMATION

1. PROJECT ______________________________

2. DATE ______________________________

3. ANALYST ______________________________

4. PRODUCT ______________________________

5. SOURCE DOCUMENTATION

____

____

____

____

METRIC WORKSHEET 11D: STANDARDS REVIEW

PHASE/REVIEW: Coding and Unit Testing
APPLICATION LEVEL: Unit

		Yes	No	NA	Unk
MO.1(3)	Are the estimated lines of source code (MLOC) for this unit 100 lines or less, excluding comments? (AMS AU.1 (2e))				
MO.1(4)	a. How many parameters are there in the calling sequence? (AMS MO.1 (5e))				
	b. How many calling sequence parameters are control variables (e.g., select an operating mode or submode, direct the sequential flow, directly influence the function of the software)?				
	c. Calculate 1-b/a and enter score.				
MO.1(5)	Is all input data passed into the unit through calling sequence parameters (i.e., no data is input through global area or input statements)? (AMS MO.1 (7e))				
MO.1(6)	Is output data passed back to the calling unit through calling sequence parameters (i.e., no data is output through global areas)?				
MO.1(7)	Is control always returned to the calling unit when execution is completed? (AMS MO.1 (9e))				
MO.1(8)	Is temporary storage (i.e., workspace reserved for intermediate or partial results) used only by this unit during execution (i.e., is not shared with other units)?				
MO.1(9)	Does this unit have a single processing objective (i.e., all processing within this unit is related to the same objective)? (AMS MO.1 (3e))				
SI.1(2)	Is the unit independent of the source of the input and the destination of the output? (AMS SI.1 (2e))				
SI.1(3)	Is the unit independent of the knowledge of prior processing? (AMS SI.1 (3e))				
SI.1(4)	Does the unit description/prologue include input, output, processing, and limitations? AMS SI.1 (4e))				
SI.1(5)	a. How many entrances into the unit? (AMS SI.1 (5e))				
	b. How many exits from the unit? (AMS SI.1 (6e))				
	c. Calculate (1/a + 1/b) * (1/2) and enter score.				
	d. If c < 1 circle N, otherwise circle Y				
SI.1(10)	Does the description of this unit identify all interfacing units and all interfacing hardware? (AMS SI.1(11e))				

METRIC WORKSHEET 11D: STANDARDS REVIEW (cont.)

PHASE/REVIEW: Coding and Unit Testing
APPLICATION LEVEL: Unit

		Yes	No	NA	Unk
SI.4(1)	Is the flow of control from top to bottom (i.e., flow of control does not jump erratically)? (AMS SI.4(1e))				
SI.4(2)	a. How many negative boolean and compound boolean expressions are used?				
	b. Calculate 1 - (a/MLOC) and enter score.				
SI.4(3)	a. How many loops (e.g., WHILE, DO/FOR, REPEAT)? (AMS SI.4(4e))				
	b. How many loops with unnatural exits (e.g., jumps out of loop, return statement)				
	c. Calculate 1 - (a/MLOC) and enter score.				
SI.4(4)	a. How many iteration loops (i.e., DO/FOR loops)? (AMS SI.4(6e))				
	b. In how many iteration loops are indices modified to alter the fundamental processing of the loop?				
	c. Calculate 1 - (b/a) and enter score.				
SI.4(5)	Is the unit free from all self-modification of code (i.e., does not alter instructions, overlays of code, etc.)? (AMS SI.4(8e))				
SI.4(6)	a. How many statement labels, excluding labels for format statements? (AMS SI.4(9e))				
	b. Calculate 1 - (a/MLOC) and enter score.				
SI.4(7)	a. What is the maximum nesting level?				
	b. Calcualte 1/a and enter score.				
SI.4(8)	a. How many branches, conditional and unconditional? (AMS SI.4(11e))				
	b. Calculate 1 - (a/MLOC) and enter score.				
SI.4(9)	a. How many data declaration statements? (AMS SI.4(12e))				
	b. How many data manipulation statements? (AMS SI.4(13e))				
	c. Calculate 1 - ((a + b)/MLOC) and enter score.				

METRIC WORKSHEET 11D: STANDARDS REVIEW (cont.)

PHASE/REVIEW: Coding and Unit Testing
APPLICATION LEVEL: Unit

		Yes	No	NA	Unk
SI.4 (10)	a. How many total data items (DD), local and global, are used? (AMS SI.4(14e))				
	b. How many data items are used locally (e.g., variables declared locally and value parameters)? (AMS SI.4(15e))				
	c. Calculate b/a and enter score.				
SI.4(11)	Calculate 1 - (DD/MLOC) and enter score.				
SI.4(12)	Does each data item have a single use (e.g., each array serves only one purpose)?				
SI.4(13)	Is this unit coded according to the required programming standard?				
SI.5(1)	a. How many data items are used as input?				
	b. Calculate 1/(1 + a) and enter score.				
SI.5(2)	a. How many data items are used for output?				
	b. How many parameters in the unit's calling sequence return output values?				
	c. Calculate b/a and enter score.				
SI.5(3)	Does the unit perform a single, nondivisable function? (AMS SI.5(4e))				
SR INPUT:	Assign a value of 1 to all Y answers and a value of 0 to all N answers. Count the total number of answers (not NA's). Total score of answers and divide by number of answers. Assign to DR.				
	TOTALS				
		DR=			

METRIC WORKSHEET 11D: STANDARDS REVIEW (2)

PHASE/REVIEW: Coding and Unit Testing
APPLICATION LEVEL: CSCI

		Yes	No	NA	Unk
MO.1(2)	Are all units coded and tested according to structural techniques?				
MO.1(3)	a. How many units in CSCI?				
	b. How many units with estimated executable lines of source code less than 100 lines?				
	c. Calculate b/NM and enter score.				
	d. If b/NM ≤ 0.5 circle N, otherwise circle Y.				
MO.1(4)	a. How many parameters are there in the calling sequence? (total from all units)				
	b. How many calling sequence parameters are control variables (e.g., select an operating mode or submode, direct the sequential flow, directly influence the function of the software)? (total from all units)				
	c. Calculate 1-(b/a) and enter score.				
MO.1(5)	a. For how many units is all input data passed into the unit through calling sequence parameters (i.e., no data is input through global areas or input statements?)				
	b. Calculate a/NM and enter score.				
	c. If a/NM < 1 circle N, otherwise circle Y.				
MO.1(6)	a. For how many units is output data passed back to the calling unit through calling sequence parameters (i.e., no data is output through global areas)?				
	b. Calculate a/NM and enter score.				
	c. If a/NM < 1 circle N, otherwise circle Y.				
MO.1(7)	a. For how many units is control always returned to the calling unit when execution is completed?				
	b. Calculate a/NM and enter score.				
	c. If a/NM < 1 circle N, otherwise circle Y.				

METRIC WORKSHEET 11D: STANDARDS REVIEW (2) (cont.)

PHASE/REVIEW: Coding and Unit Testing
APPLICATION LEVEL: CSCI

		Yes	No	NA	Unk
MO.1(8)	a. For how many units is temporary storage (i.e., workspace reserved for immediate or partial results) used only by the unit during execution (i.e., is not shared with other units)? b. Calculate a/NM and enter score. c. If a/NM < 1 circle N, otherwise circle Y.				
MO.1(9)	a. How many units have a single processing objective (i.e., all processing within the unit is related to the same objective)? b. Calculate a/NM and enter score. c. If a/NM ≤ 0.5 circle N, otherwise circle Y.				
SI.1(2)	a. How many units are independent of the source of the input and the destination of the output? b. Calculate a/NM and enter score. c. If a/NM ≤ 0.5 circle N, otherwise circle Y.				
SI.1(3)	a. How many units are independent of knowledge of prior processing? b. Calculate a/NM and enter score. c. If a/NM ≤ 0.5 circle N, otherwise circle Y.				
SI.1(4)	a. For how many units does the unit description/prologue include input, output, processing, and limitations? b. Calculate a/NM and enter score. c. If a/NM ≤ 0.5 circle N, otherwise circle Y.				
SI.1(5)	a. How many units with answer of Y in W/S 11A (i.e., number of entrances = 1, number of exits = 1)? b. Calculate a/NM and enter score. c. If a/NM < 1 circle N, otherwise circle Y.				
SI.1(7)	a. How many unique data items are in common blocks? b. How many unique common blocks? c. Calculate b/a and enter score.				
SI.1(10)	Do all descriptions of all units identify all interfacing units and all interfacing hardware?				

METRIC WORKSHEET 11D: STANDARDS REVIEW (2) (cont.)

PHASE/REVIEW: Coding and Unit Testing
APPLICATION LEVEL: CSCI

		Yes	No	NA	Unk
SI.2(1)	a. How many units are implemented in a structured language or using a preprocessor?				
	b. Calulate a/NM and enter score.				
	c. If a/NM ≤ 0.5 circle N, otherwise circle Y.				
SI.4(1)	a. For how many units is the flow of control from top to bottom (i.e., flow of control does not jump erratically)?				
	b. Calculate a/NM and enter score.				
	c. If a/NM ≤ 0.5 circle N, otherwise circle Y.				
SI.4(2)	a. How many executable lines of code (LOC) in this CSCI?				
	b. How many negative boolean and compound boolean expressions are used ? (total from all units)				
	c. Calculate 1 - (b/LOC) and enter score.				
SI.4(3)	a. How many loops (e.g., WHILE, DO/FOR, REPEAT)? (total from all units)				
	b. How many loops with unnatural exits (e.g., jumps out of loop, return statement)? (total from all units)				
	c. Calculate 1 - (b/a) and enter score.				
SI.4(4)	a. How many iteration loops (i.e., DO/FOR loops)? (total from all units)				
	b. In how many iteration loops are indices modified to alter fundamental proccessing of the loop? (total from all units)				
	c. Calculate 1 - (b/a) and enter score.				

METRIC WORKSHEET 11D: STANDARDS REVIEW (2) (cont.)

PHASE/REVIEW: Coding and Unit Testing
APPLICATION LEVEL: CSCI

		Yes	No	NA	Unk
SI.4(5)	a. How many units free from all self-modification of code (i.e., does not alter instructions, overlays of code, etc.)?				
	b. Calculate a/NM and enter score.				
	c. If a/NM ≤ 0.5 circle N, otherwise circle Y.				
SI.4(6)	a. How many statement labels, excluding labels for format statements? (total from all units)				
	b. Calculate 1 - (a/LOC) and enter score.				
SI.4(7)	a. What is the maximum nesting level? (total from all units)				
	b. Calculate 1/a and enter score.				
SI.4(8)	a. How many branches, conditional and unconditional? (total from all units)				
	b. Calculate 1-(a/LOC) and enter score.				
SI.4(9)	a. How many declaration statements? (total from all units)				
	b. How many data manipulation statements? (total from all units)				
	c. Calculate 1 - ((a + b) / LOC) and enter score.				
SI.4(10)	a. How many total data items (DD), local and global, are used? (total from all units)				
	b. How many data items are used locally (e.g., variables declared locally and value parameters)? (total from all units)				
	c. Calculate b/a and enter score.				

METRIC WORKSHEET 11D: STANDARDS REVIEW (2) (cont.)

PHASE/REVIEW: Coding and Unit Testing
APPLICATION LEVEL: CSCI

		Yes	No	NA	Unk
SI.4(11)	a. Calculate DD/LOC and enter score. (total from all units)				
SI.4(12)	a. For how many units does each data item have a single use (e.g., each array serves only one purpose)?				
	b. Calculate a/NM and enter score.				
	c. If a/NM < 1 circle N, otherwise circle Y.				
SI.4(13)	a. How many units are coded according to the required programming standard?				
	b. Calculate a/NM and enter score.				
	c. If a/NM ≤ 0.5 circle N, otherwise circle Y.				
SI.4(14)	Is repeated and redundant code avoided (e.g., through utilizing macros, procedures and functions)?				
SI.5(1)	a. How many data items are used as input? (total from all units)				
	b. Calculate 1/(1 + a) and enter score.				
SI.5(2)	a. How many data items are used as output (total from all units)?				
	b. How many parameters in the units' calling sequence return output values (total from all units)?				
	c. Calculate b/a and enter score.				
SI.5(3)	a. How many units perform a single, non-divisible function?				
	b. Calculate a/NM and enter score.				
	c. If a/NM ≤ 0.5 circle N, otherwise circle Y.				
SR INPUT:	In "VALUE" column, asign 1 to all "Y" answers, 0 to all "N" answers, and enter score for all elements (e.g., SI.4(11), SI.5(1)) for which Y/N option not provided. Sum entries in "VALUE" column, divide by number of answers, and subtract quotient from 1. Assign result to DF.				
	TOTALS				
		DR=			

PROCEDURE NO. 12

1. Title: Execution Time (ET)

2. Estimation Parameter Supported: Failure Rate

3. Objectives: Execution times are used in conjunction with Software Discrepancy Reports to obtain failure rates. The number of failures per time period is the basic reliability measure used in this Guidebook.

4. Overview: Execution time is the interval during which the central processing unit (CPU) of the computer executes the program. Two measures are suggested. One is the actual CPU execution time. The other is computer operation time. Execution time will generally refer to each. Computer Operation time will be the default.

5. Assumptions/Constraints: Execution time cannot be directly compared between machines of different word length. Significant differences in machine architectures may make it impossible to compare execution times accurately. It is assumed, using the method given in item 10, that comparisons of sufficient accuracy can be made. The unit of time used in both measures of execution time is hours.

6. Limitations: The accuracy of execution time estimates will be affected by the type of timing device available in the system under test. Since not all operating systems are capable of sufficiently precise timing, statistical measures derived from the Execution Time measurements must not assume greater precision than is actually available.

7. Applicability: Execution time may be obtained during CSC integration and testing, CSCI level testing, system integration and testing, operational test and evaluation and operations. Since it is used in conjunction with Software Discrepancy Reports, it should be obtained during the relevant reporting periods.

8. Required Inputs: Execution times are typically obtained from software operating system reports or test reports.

9. Required Tools: No special tools, other than those provided by the operating system, will be required.

10. Data Collection Procedures: Execution time is obtained from operating system records,or tester's logs, which typically report the execution time for each program or project on a run basis, as well as daily, weekly, or monthly totals. Where operating system reports are not available, execution time may be expressed in computer time, the time during which the computer (as contrasted with the CPU) executes the program.

    In cases in which execution time is not available, it may be estimated from total computer time with one of the following methods:

    a. Running a benchmark HOL program on a mainframe on which execution time will be reported, and then running the same test case on the target computer.

    b. Running a program on the target computer in a manner that will eliminate or minimize disk access (e.g., by putting data in memory) and output operations, thus obtaining essentially an execution time measurement, and then running the same test case in the normal manner.

c. Counting the number of I/O operations involved in a program and computing the nominal time for these from the computer instruction manual.

When comparisons are made between programs running on different computers, it is necessary to normalize execution time for word length and execution speed. Raw execution time is divided by the number of bits executed per second, which is obtained by multiplying computer word length in bits by the number of instructions per second. This figure may be modified in the case of machines which use more than one word for instructions, or which can have more than one instruction per word.

11. Outputs: Execution times are reported for use in calculation of failure rates. This guidebook recommends the use of computer operation time since it is more generally available.

12. Interpretation of Results: As noted in paragraph 10, raw execution times may be misleading, because of variations in computer word lengths, speed, and timing mechanisms employed. Because of the importance of the Failure Rate in validation of software reliability metrics, it will be essential to obtain accurate and reliable measures of Execution Time.

13. Reporting. Execution times are reported for use in Failure Rate measurements.

14. Forms: Worksheet 12 is prepared manually from data obtained from operating system outputs or tester's logs.

15. Instructions: During formal testing, it is important to record not only the problems encountered (see Worksheet 6) but also the amount of testing performed. This data allows calculation of the failure rate being experienced during testing. Worksheet 12 is provided to facilitate the required record keeping. Each individual Tester should complete these worksheets. Each individual test run should be recorded, the date it was run, a reference to a test plan or procedure if appropriate, reference to a discrepancy report if there was a problem encountered during the test run, and the execution time of the test run. Note, successful test runs as well as unsuccessful test runs should be recorded with execution time. Reference to a discrepancy report is only made if a problem is encountered.

    Execution time should be recorded in computer operation hours (wall clock time of run) and/or in CPU hours if CPU execution time is available. The measure of time should be indicated.

16. Potential/Plans for Automation: This metric is generated automatically by most operating systems.

17. Remarks: As noted, Execution Time cannot be compared directly between systems running on different machines. This problem can be expected to increase as specialized machine architectures are used (e.g., data base machines).

WORKSHEET 12

TEST LOG

CSCI NAME______________________________

TEST NAME______________________________

TEST LOCATION__________________________

PERSON COMPLETING LOG__________________

WITNESS(ES)____________________________

HARDWARE CONFIGURATION_________________

SOFTWARE CONFIGURATION_________________

DATE___________________________________

EXECUTION TIME_________________________

TIME	EVENT

PROCEDURE NO. 13

1. Title: Failure Rate (F)

2. Prediction or Estimation Parameter Supported: Failure Rate

3. Objectives: The Failure Rate is the ultimate measure of software reliability used in this guidebook. It represents the ground truth, which the metrics, in combination, are attempting to approximate. Failure Rate provides a measure of system reliability. Mission software failure probability is the product of software failure rate and mission duration; mission software reliability is 1 - mission software failure probability.

4. Overview: This metric is obtained by dividing the number of failures reported over a standard time period. Time-stamped software discrepancy reports are used to provide a count of system failures during the stated time period. The reports also typically indicate the module or CSCI with which they are associated.

5. Assumptions/Constraints: It is assumed that software discrepancy reports will provide an accurate measure of the failure rate of software over time. Preparation of discrepancy reports may not follow similar procedures on different projects. Even on the same project, as a deadline approaches, programmers may tend to feel that there is not time to prepare reports for failures that they perceive as minor, even though they might have prepared them at earlier times. The assumption, then, is that the programming environment is disciplined enough to enforce a consistent error reporting procedure. Automation of error reporting, if feasible, would help to increase consistency. Nevertheless, since the failure rate is essential for testing and validating all other metrics, it will be necessary to enforce consistency in the collection of data for this purpose.

6. Limitations: As noted in the preceding paragraph, consistency in data collection is assumed.

7. Applicability: This metric is obtained during operational tests and later operations and maintenance. It serves to validate predictions and estimates obtained during preceding phases of the software development cycle.

8. Required Inputs: Software discrepancy reports are used to measure the number of failures over time. The operating system is used to track the operation time.
9. Required Tools: None.

10. Data Collection Procedures: The software discrepancy reports are counted. In many cases, reports are maintained on disk, so that counts will be immediately available. Time stamps and modules are used to permit identification of reports with designated time periods and software segments. The required metric is obtained by dividing the number of discrepancy reports by the number of hours of computer operation time to obtain the failure rate for any designated unit, CSCI,or system. The average failure rate during testing, FT1, is calculated by taking the number of discrepancy reports recorded and dividing by the total amount of test time recorded. This average can be calculated anytime during testing and represents the current average failure rate. When calculated, it is based on the current total number of discrepancy reports recorded and the current total amount of test operation time expended. It is expected that the failure rate will vary widely depending on when it is computed. For more consistent results, average failure rates should be calculated for each software test phase: CSC Integration and Testing, CSCI Testing; and, if required, for each system test phase: Systems Integration and Testing, and Operational Testing and Evaluation.

The failure rate at end of test, FT2, is calculated by taking the number of discrepancy reports recorded during the last three test periods of CSCI Testing and dividing by the amount of test time recorded during these last three periods. This failure rate can be updated at the end of System Integration and test and at the end of Operational Test and Evaluation. A test period is defined as a test interval or session with specific test objectives. A test period could be a test run, a day or a month.

11. Outputs: The basic statistic output by this procedure is the failure rate. Since all metrics have been stated in terms of this rate, no further transformation should be required.

12. Interpretation of Results: The failure rate is interpreted as the primary measure of software reliability.

13. Reporting: Failure Rate is a basic measure of software quality and may be specified by the sponsoring agency or user. It is, therefore, essential to report failure rates to the project manager, to provide evidence that contractual requirements are being fulfilled.

14. Forms: Failure rates are to be reported for each module, CSCI, and system as part of the normal reporting procedure. Metric Worksheet 13 can be used.

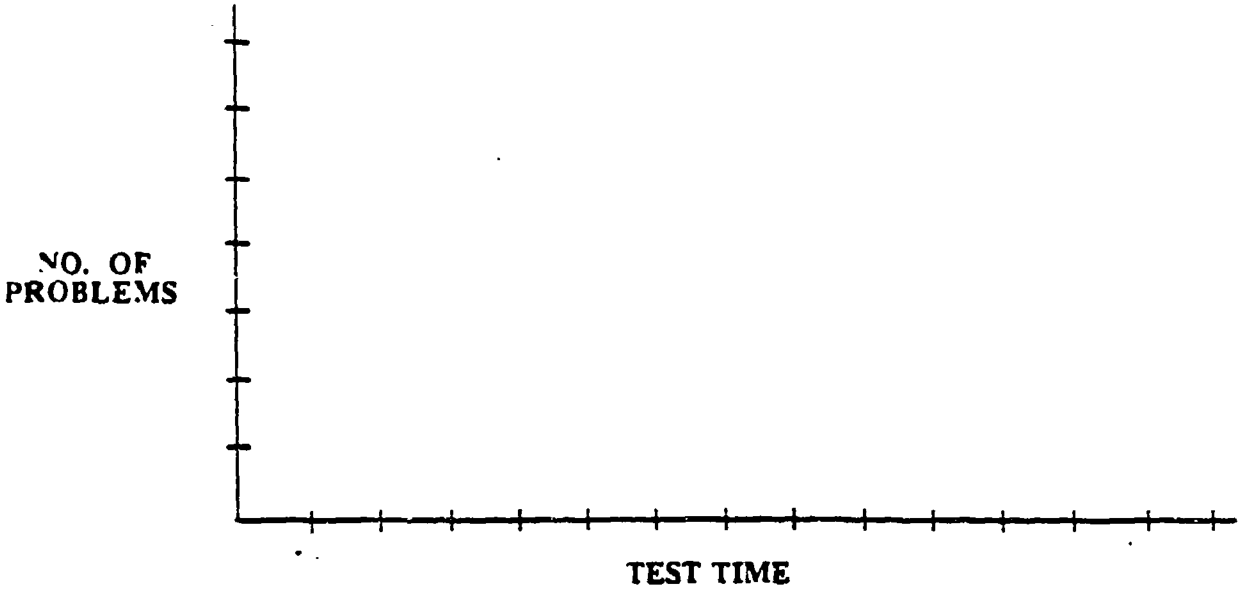
15. Instructions: Worksheet 13 can be used to track testing progress. The units of time on the horizontal axis should be chosen to represent the test phase of the project. The number of problems recorded each test period should be plotted to facilitate observation of the trend in failure rate.

    Worksheet 13 also supports calculation of the average failure rate during test (F_{T1}) and the failure rate at end of test (F_{T2}).

16. Potential/Plans for Automation: An automated procedure will provide an objective record of unit failures, although it is not likely that it will be able to provide complete information concerning the reasons for errors. In addition, it may not be able to detect failures in which outputs are not within required tolerances. In short, not all software failures are detectable by an automated system. Automation will be most valuable in maintaining error reports on-line, in an accessible form, for review by the project manager and quality control personnel.

17. Remarks: An accurate measure of failure rate is essential to the success of efforts to obtain appropriate metrics.

WORKSHEET 13
FAILURE RATE TREND



FAILURE RATE CALCULATION

AVERAGE FAILURE RATE DURING TEST:

$$F_{T1} = \frac{\text{Total number of Discrepancy Reports during Test}}{\text{Total Test Time}}$$

= ____________ / ____________ = ____________

Failure Rate at end of Test:

$$F_{T2} = \frac{\text{No. of Discrepancy Reports during last 3 test periods}}{\text{Total Test Time during last 3 test periods}}$$

= ____________ / ____________ = ____________

PROCEDURE NO. 14

1. Title: Test Effort (TE)

2. Prediction or Estimation Parameter Supported: Test Environment

3. Objectives: Test Effort is a measure of the quantity of testing to be performed. Three alternative measures are available. One is determined by the number of person days expended. One is determined by the amount of funds allocated to testing. One is determined by the number of calendar days expended during each phase of testing, normalized by the total number of days (or hours) for the development effort.

4. Overview: Estimates of the number of hours to be expended in testing are used during the early phases of the project. As actual numbers of hours become available, they are used to correct the early estimates.

   It is important to note that the amount of test effort expended is impacted by the testing techniques selected for the test program. See Data Collection Procedure No. 16, Test Methodology, for guidance on selecting testing techniques with the goal of reducing test effort.

5. Assumptions/Constraints: The Test Effort measurement requires access to labor hour data for a project and a work breakdown structure accounting system that delineates labor expended during testing. It is assumed that accurate figures for the hours of testing and total hours for the project are available. Because reported hours are not always accurate (e.g., because of unpaid overtime), some inaccuracy may appear in the reported hours.

6. Limitations: A measure of formal program testing may not include all testing performed. Typically, informal tests are performed at all levels throughout the development cycle. If these informal tests are frequent, and are not reported as such, the metric may be somewhat distorted, since the time for formal testing may be reduced without reducing the reliability of the software.

7. Applicability: Estimates of the amount of testing to be performed may be obtained throughout the software development cycle.

8. Required Inputs: For measurement of the amount of testing, job records from the software development project may be used. At earlier phases of the project, estimates of time to be spent in testing will be employed.

9. Required Tools: This factor will obtained manually from management reports.

10. Data Collection Procedures: Periodic project reports will be reviewed to obtain data concerning hours expended on software tests.

11. Outputs: The number of hours (or days) of testing, divided by the total number of hours (or days) in the development, will be used in computing this metric (AT). The value of the metric AT is used to determine the multiplier, TE.

12. Interpretation of Results: The amount of testing should provide an indication of software reliability, in that more thoroughly tested software is likely to contain fewer remaining errors. The effect of this measure could be balanced against the difficulty or complexity of the application, but no effective measure of the difficulty is available.

13. Reporting: A monthly report of the amount of tes:ing would be appropriate, and would provide the project manager with a continuing record of effort expended on tests.

14. Forms: The metric worksheet 14 ca:: be used.

15. Instructions: Wo:ksheet 14 is provided to support calculation of the Test Effort Metric. The three alternative calculations are described in that worksheet. Inputs to these calculations come from management reports which track resource and budget expenditures and schedule.

Test effort represents the amount of effort applied to software testing. Three alternatives are available in Worksheet 6B for evaluation of test effort. Each evaluates the percentage of effort, budget or schedule devoted to testing and compares that with a guideline of 40%.

a. First choice: Labor Hours (Alternative 1)
b. Second choice: Dollars (Alternative 2)
c. Last choice: Schedule (Alternative 3)

A discussion of s:veral possible approaches to determine test completion criteria based on lab r hours follows. The third approach given derives from the experiment and results documented in Volume 1.

Approach 1 - Test Until a Method Is Exhausted

An example of this approach might be the requirement to test until all logical paths in the software have been executed and no errors remain. An alternative requirement might be to test the software until all boundary-value cases have proved to be error free. This technique has limits of effectiveness: it is not helpful in a test phase in which the methodology is not applicable, such as an operational test. Further, it is a subjective criterion, because there is no way to determine that the methodology is applied rigorously, nor that it is necessarily the most appropriate methodology.

Approach 2 - Error Prediction Models

Several error-prediction models, or software reliability models are available. There have been a number of such models proposed in the technical literature. Once of the best summaries is "A Guidebook for Software Reliability Assessment", RADC TR 83-176.

Some models require testing the software for a length of time and recording the elapsed time between detection of successive errors. Other models require recording computer execution time between detected errors.

Approach 3 - Stopping Rules

Discrete rules for when sufficient testing will have been accomplished are shown to be an effective method of controlling test effort. Table TS301-1 contains stopping rules which were adopted for the software testing experiments and studies described in V lume 1. They were validated by actual testing, are largely conservative in effect and are b recommended.

WORKSHEET 14
TEST EFFORT (TE)

ALTERNATIVE 1: LABOR HOURS

a. Budget in terms of labor hours for the software testing effort ____________

b. Budget for the entire software development effort in terms of labor hours ____________

c. Calculate a/b and enter score ____________

d. Calculate .40/c and enter score TE = ____________

ALTERNATIVE 2: DOLLARS

a. Budget in terms of dollars for the software testing effort. ____________

b. Budget for the entire software development effort in terms of dollars ____________

c. Calculate a/b and enter score ____________

d. Calculate .40/c and enter score TE = ____________

ALTERNATIVE 3: SCHEDULE

a. Schedule for software testing in terms of work days ____________

b. Schedule for entire software development in terms of work days ____________

c. Calculate a/b and enter score ____________

d. Calculate .40/c and enter score TE = ____________

TABLE TS301-1 STOPPING RULES

TEST TECHNIQUE	STOPPING RULE	TEST LEVEL	
		Unit	CSC*
Branch Testing	100% of branches executed (with a minimum of 2 traversals per branch) *and* MTTF = 10 input cases. Not to exceed X hours.	X = 29	X = 58
Code Review	All required aspects of the method have been evaluated using SDDL where possible, manually where not. Not to exceed X hours.	X = 8	X = 16
Functional Testing	All test procedures executed. Not to exceed X hours.	X = 16	X = 32

Random Testing	Minimum number, Y, samples from input space executed, *and* MTTF = 10 input cases. Not to exceed X hours.	X = 22
		Y = 25
			X = 44
			Y = 50

Error & Anomaly Detection	All required aspects of the method have been evaluated, using automated tool where possible, manually where not. Not to exceed X hours.	X = 6	X = 12
Structure Analysis	All required aspects of the method have been evaluated, using automated tool where possible, manually where not. Not to exceed X hours.	X = 4	X = 8

Note: Unit test level stopping rules were used for CSC testing, due to budget constraints.

Test effort represents the time testers take to reach the stopping rules for each testing technique. As such, test effort factors heavily in the testing technique selection strategies in Data Collection Procedure No. 16, Test Method. Key contributions of the test effort data from the experiments conducted are:

a. The single unit and CSC technique effort results showed static techniques took less time than the dynamic ones, and technique pairs of two static techniques also took less time than other pairs of techniques.

b. Estimates for the average effort at the unit and CSC level, not including driver development and test environment setup, can be made from the estimation data in Table TS301-1.

c. The driver development and test environment setup depend greatly upon the code under test and the tools used, respectively. Thus the data provided in Table TS301-1 for estimation purposes needs to be adjusted with regard to code under test characteristics, and tools and environment used for the testing.

d. The testing techniques took approximately the same relative amount of time across samples from the two different projects in the study, but the samples with the greater size and complexity took more actual time.

Individually, error/anomaly detection and structure analysis require the least time, then code review. Random testing and functional testing are next. Branch testing uses the most time. These rankings are based on the experiment data.

Recommended pairs of testing techniques are listed below in order of increasing effort necessary to apply them. Pair-combinations that are not listed here are considered inapplicable for selection based on the criteria of test effort alone.

a. Error/Anomaly Detection - Structure Analysis
b. Code Review - Error/Anomaly Detection
c. Code Review - Structure Analysis
d. Code Review - Functional Testing
e. Code Review - Random Testing

16. Potential/Plans for Automation: This metric will be extracted manually from management reports.

17. Remarks: The project manager should keep accurate records of the time spent in software testing. In some instances, testing is not clearly broken out as a separate project task. Alternative methods for collecting this measure are by using funding instead of time.

PROCEDURE NO. 15

1. Title: Test Methodology (TM)

2. Prediction or Estimation Parameter Supported: Test Environment

3. Objectives: This metric is a measure or assessment of the test methodology. It is based on the techniques and tools employed.

4. Overview: This Data Collection Procedure provides an approach for identifying what testing techniques should be employed based on considerations that will reduce test effort (TE), test coverage (TC), and various types of software errors which can be encountered during the testing process. Then the recommended testing techniques are scored versus the test methodology actually employed in order to assess the thoroughness of the test program.

5. Assumption/Constraints: Determination of this metric requires that the number of test tools be counted. This assumes that it is possible to count different tools and techniques meaningfully, although in some instances a tool may have several functions, or a number of tools may be integrated into a comprehensive testing environment. It also assumes that the distinction between software test tools and other support software (such as editors) is clear-cut.

6. Limitations: As noted, the use to test tools and techniques relies on several elements that may not be well-defined in particular applications.

7. Applicability: Information concerning projected use of test tools and techniques will be available at the requirements phase, as part of the Test Plan. The projected use of tools and techniques will be included in the Software Development Plan also. Reports on actual use of tools will become available during test and evaluation.

8. Required Inputs: Test Plans, Test Procedures, Software Development Plan.

9. Required Tools: Information concerning the use of tools and techniques will be obtained manually from project reports, as noted.

10. Data Collection Procedures: Task Section 200 should be used to develop a list of tools and techniques that should be used. Each one used, confirmed by observing testing or reviewing documents, would be checked off.

11. Outputs: Output from this procedure will be the reported number of test tools and techniques were used (TU) and the total recommended (TT).

12. Interpretation of Results: The use of test tools and techniques is expected to produce a more effective and objective testing methodology, which should be reflected in greater system reliability.

13. Reporting: Information concerning the use of test tools and techniques should be reported back to the project monitor, as well as the project manager, to ensure that there is understanding of the role of tools in the software development effort.

14. Forms: Use Data Collection Worksheets 7A through 7C.

WORKSHEET 7A

TEST TECHNIQUE SELECTION BASED ON TEST EFFECTIVENESS/TEST COVERAGE (UNIT LEVEL)

TEST TECHNIQUE	SELECTED (X)	
	Single	Paired
Error/Anomaly Detection		
Code Review		
Branch Testing		
Functional Testing		
Structure Analysis		
Random Testing		

WORKSHEET 7B

TEST TECHNIQUE SELECTION BASED ON TEST EFFECTIVENESS (CSC LEVEL)

TEST TECHNIQUE	SELECTED (X)	
	Single	Paired
Error/Anomaly Detection		
Code Review		
Branch Testing		
Functional Testing		
Structure Analysis		
Random Testing		

WORKSHEET 7C

TEST TECHNIQUE SELECTION FOR TEST EFFECTIVENESS, TEST COVERAGE AND ERROR CATEGORY

SOFTWARE TEST TECHNIQUES		TEST EFFECTIVENESS		TEST COVERAGE		ERROR CATEGORY 7E	NOTES/ COMMENTS
		7A	7B	7A	7B		
STATIC	CODE REVIEWS						
	ERROR/ANOMALY DETECTION						
	STRUCTURE ANALYSIS/DOC.						
DYNAMIC	RANDOM TESTING						
	FUNCTIONAL TESTING						
	BRANCH TESTING						

Enter an 'X' opposite the testing texhniques(s) selected from Worksheets 7A, 7B, and 7E.

15.. Instructions: Task Section 200 and RADC TRF 84-53 provides a methodology for identifying the appropriate test techniques to use during a software development project. The procedures should be followed and the recommended techniques should be documented in Worksheet 15. Then, during testing, Worksheet 15 can be used as a checklist to assess which techniques are actually used to test the software. The Test Plan, Specifications and Procedures as well as the Software Development Plan should be reviewed also.

The Test Methodology metric, TM, then is based on the ratio of the applied techniques (TU) to the recommended techniques and tools (TT).

16. Potential/Plans for Automation: This metric is essentially a description of test management, which is extracted manually from project documentation, rather than through the use of automated tools.

WORKSHEET 15

TEST METHODOLOGY CHECKLIST

LIST TECHNIQUES AND TOOLS RECOMMENDED IN RADC TR 88-XX	CHECK THOSE THAT ARE ACTUALLY USED
TOTAL NUMBER RECOMMENDED: TT=	TOTAL NUMBER USED: TU=

PROCEDURE NO. 16

1. Title: Test Coverage (TC)

2. Prediction or Estimation Parameter Supported: Test Environment

3. Objectives: Test Coverage is a measure of the thoroughness of testing in terms of how thoroughly the code was executed during dynamic testing of the system.

4. Overview: Using available test tools, a count is taken which assesses coverage (VS). This coverage can be assessed during unit testing looking at branches executed, at integration testing looking at units and interfaces tested, or at system testing looking at requirements tested.

   The amount of test coverage that can be attained at lower test levels is impacted by the testing techniques selected for the test program. See Data Collection procedure no. 15, Testing Methodology, for guidance on selecting testing techniques with the goal of increasing test coverage.

   The following data will be obtained during the indicated test phases:

   a. Unit Test

      (1). Percent of executable lines of code exercised during all unit tests

      (2). Percent of branches exercised during all unit tests

   b. Integration and test

      (1). Percent of modules exercised during implementation and test

      (2). Percent of all interfaces exercised during implementation and test

   c. Demonstration/Operational Test and Evaluation

      (1). Percent of functions exercised

      (2). Percent of user scenarios exercised

      (3). Percent of I/0 options exercised

5. Assumptions/Constraints: Not all branches and calls are actually equal in determining software reliability. For example, a well-designed system may include a large number of error procedures which are never called during normal system operation. Some portions of code may be used only when hardware or software failures are encountered. It may be difficult to exercise these portions of code during system tests.

6. Limitations: It should be noted that the exercise of a portion of code does not, in itself, provide any guarantee that the code will perform correctly over the full range of program variables. At best, it provides evidence that the code is capable of functioning for some value of the variables that it uses.

7. Applicability: This metric may be obtained during unit tests, integration and testing, and demonstration and operational test and evaluation.

8. Required Inputs: Test programs normally provide data concerning the extent of testing, as noted above. The following data elements will be required:

   TP = Total number execution branches
   PT = Number of execution branches tested
   TI = Total number of inputs
   IT = Number of inputs tested
   NM = Total number of units
   MT = Number of units tested
   TC = Total number of interfaces
   CT = Number of interfaces tested
   NR = Total number of requirements
   RT = Number of requirements tested

9. Required Tools: Appropriate test tools are available for exercising software systems and for obtaining required inputs. It will be necessary to identify appropriate test tools for specific systems to be tested.

10. Data Collection Procedures: The inputs described in paragraph 8 above are to be extracted during the project phases in paragraph 4. These are combined using the formula in Task 201 to obtain the Test Coverage metric, TC.

11. Outputs: The Test Coverage metric will be used in the computation of the Test Environment metric.

12. Interpretation of Results: A complete testing procedure would exercise all possible combinations of paths through the software system, using data for the full range of permissible and impermissible (erroneous) values. Such tests of any reasonably complex system become expensive, because of the enormous number of combinations of values and branches to be exercised. The Test Coverage metric must therefore be interpreted in terms of the ultimate user of the system, the cost of failures, and mechanisms for recovery. From the point of view of cost-effectiveness, full tests of a system may not be preferable to less expensive partial tests, providing that the cost of failure is not excessive.

13. Reporting: Reports should indicate serious failures in the testing process, where tests have failed to cover significant portions of the software system. For the project manager, such reports are valuable.

14. Forms: Worksheet 16 is for each of the three metrics.

15. Instructions: Collecting data to assess how thoroughly a software system is tested is difficult unless:

    (1) A Requirements/Test Matrix has been developed.

    (2) A tool is used during testing which instruments the code and reports coverage data based on test case execution.

Worksheet 16 assumes one or both these data sources are available. Values for NM and NR were collected on other worksheets. Data Collected during unit testing can also be collected during integration testing and system testing. The unit and CSC level data should be accumulated and averaged at the CSCI level. Data collected at the Integration Test level can be collected during System Test also.

Test coverage, in terms of software operation is the total number of branches executed from ali tests. As such, test coverage factors heavily in testing techniques selection strategies.

The main factors affecting test coverage that can be attained appear to be the software being tested (e.g. its number of branches) and the test technique applied. Branch testing achieved the highest coverage at unit and CSC levels, not surprisingly, since its stopping rule calls for a high branch coverage. Functional testing and random testing achieved virtually the same branch coverage at the unit level, but functional testing slightly outperformed random testing at the CSC level. Branch coverage decreased for all techniques at the CSC level.

Experimental data for single testing techniques show that it is an effective strategy to combine testing techniques to increase test coverage. (Static techniques cannot be included here because their application does not involve executing the code, and therefore executing branches.) The branch/functional technique pair obtained the highest coverage, followed by branch/random testing, and then functional/random testing.

Another interesting view of coverage can be termed "coverage efficiency", or the comparison of coverage reached by a technique relative to the time taken to reach that coverage. In comparing the descriptive paired technique data for effort and coverage, it is seen that the coverage is inversely related to the effort: the technique-pair that took the longest to apply attained the best branch coverage, and vice versa.

16. Potential/Plans for Automation: Test Coverage can be computed automatically through the use of software test tools.

17. Remarks: Test Coverage is an important metric for evaluating the quality of testing that has been applied to the software system. It provides a measure of the degree of confidence that the manager can have in the results of testing.

WORKSHEET 16
TEST COVERAGE

DURING UNIT TEST (FOR EACH CSC OR UNIT):

TOTAL NUMBER OF EXECUTION BRANCHES: TP = ____________
TOTAL NUMBER OF EXECUTION BRANCHES TESTED: PI = ____________
TOTAL NUMBER OF INPUTS: TI = ____________
TOTAL NUMBER OF INPUTS TESTED: IT = ____________

DURING INTEGRATION TESTING (FOR EACH CSCI):

TOTAL NUMBER OF UNITS: NM = ____________
TOTAL NUMBER OF UNITS TESTED: TM = ____________
TOTAL NUMBER OF INTERFACES: TC = ____________
TOTAL NUMBER OF INTERFACES TESTED: CT = ____________

DURING SYSTEM TESTING:

TOTAL NUMBER OF REQUIREMENTS: NR = ____________
TOTAL NUMBER OF REQUIREMENTS TESTED: RT = ____________

PROCEDURE NO. 17

1. Title: Exception Frequency (EV)

2. Prediction or Estimation Parameter Supported: Operating Environment

3. Objectives: This metric represents the view that the greater the variability of inputs to the program, the more likely an unanticipated input will be encountered and the program will fail.

4. Overview: A measure of program variability (EV) will be obtained through a count of exception conditions that occur over a period of time. Hardware monitors will provide the required data. The value of EV may be represented as:

$EV = .1 + 4.5EC$

where EC is a count of the number of exceptions encountered in an hour.

5. Assumptions/Constraints: There has not been sufficient testing of variability as a possible factor in software system failures. In the form described here, however, it is plausible to suppose that the number of minor and recoverable problems, as measured by the number of exception conditions, is proportional to the number of major failures, and may be used during system implementation and test to estimate failure rate.

6. Limitations: This metric is derived from hardware and software exception reports, which are normally generated by the operating system. It will, therefore, provide the basis for estimating failure rates to be expected during operation of the system. It will not be available until initial system test.

7. Applicability: Exception Frequency is determined during the coding phase, testing, and O&M.

8. Required Inputs: Exception reporting is obtained from system monitors which generate records of hardware and software failures.

9. Required Tools: The appropriate system capabilities for monitoring, reporting, and summarizing exceptions must be available for use.

10. Data Collection Procedures: Records of hardware and software exceptions are obtained, from which a count of exceptions over a series of time periods will be prepared and averaged.

11. Outputs: The results are reported as EV, as defined above. This represents a normalized figure for the number of exceptions per time period. The normalization permits the value of EV to represent the degree of increase in expected failures, reflecting the frequency of exceptions.

12. Interpretation of Results: Hardware failures are likely to account for a substantial number of exceptions. A faulty disk or tape, or faulty components in the drive mechanisms for their supporting equipment, can generate large numbers of exceptions over a period of time. For this reason, it will be important to provide some

explanation of the causes of the exceptions, to permit a proper interpretation of abnormally high exception rates.

13. Reporting: Exception frequencies should be reported back to the project monitor in cases in which excessive or anomalous values are encountered. This metric is valuable in estimating potential failure rates, by identifying specific modules or functions for which the anomaly rate is high.

14. Forms: Exception rates are based on data collected by hardware and software monitors and reported by operating system functions. This information can be entered into report forms to obtain the required value for EV.

15. Potential/Plans for Automation: A system for the collection of software metrics could include required functions for obtaining exception rates and for transforming them into the specified outputs.

16. Remarks: The exception rate appears to provide the basis for a highly accurate estimate of the failure rate to be expected during later system operations.

PROCEDURE NO. 18

1. Title: Workload (EW).

2. Prediction or Estimation Parameter Supported: Operating Environment.

3. Objectives: The Workload metric represents an estimate of the workload of the system. It is thought to be more likely that a specific task will fail in a heavily loaded system than in a lightly loaded system.

4. Overview: One measure of workload is the amount of overhead being utilized. It represents how much I/O, system calls, swapping/paging,etc is going on. In most mainframes this measure is reported by the operating system. The EW is obtained by calculating the ratio of execution time to execution time minus overhead.

5. Assumptions/Constraints: To obtain a metric which will predict reliability, it will be necessary to obtain a figure for overhead which is typical of the times when there is significant activity. Overall averages for workload will have little predictive value if they include long periods when the computer system is completely idle. For that reason, the average should be computed during peak usage.

6. Limitations: Estimates of workload should accurately reflect conditions in the operating environment. The possibility of rapid system degradation under conditions of heavy overload should be considered. Another point for consideration is possibility that system reliability will degrade -- i.e., the failure rate will increase -- in a non-linear fashion as the workload increases. There may be no failures attributable to system overload while the workload is less than, say, 95 percent; at this point, the failure rate begins to increase dramatically. The manner in which this metric is calculated assumes a linear relationship between workload and failure rate.

7. Applicability: A measure of workload can be determined during the coding phase, testing, and O&M. We are attempting to estimate what the workload will be like in operation. Stress tests, during which workload is deliberately kept at a high level, can be used to measure the effect.

8. Required Inputs: Computation of this metric will require data concerning total run time and overhead time.

9. Required Tools: Information required for this metric is normally available through the system monitor.

10. Data Collection Procedures: As data concerning overhead and total run time become available, the ratio is computed and reported. Use Data Collection Worksheet 18.

    Often, the ratio appears in accounting information produced by the system monitor.

11. Outputs: The ratio (EW) is reported as an output to the computation of the Operating Environment metric.

12. Interpretation of Results: In general, it may be expected that system performance will degrade rapidly as the CPU approaches saturation. The problem for consideration will be the extent to which software degrades in a nondestructive manner, maintaining as many mission-critical functions as possible. Outputs from this metric should be

useful in identifying a point at which degraded performance begins. Typically, Government specifications required that no more than 75 percent of system capacity be used, i.e., that there is a 25 percent margin for error, for mission-critical systems.

13. Reporting: Busy time or workload should be reported with other management data concerning resources use.

14. Forms: Workload is obtained from system management records, which are normally generated automatically by the operating system.

15. Instructions: Three data items are required to derive the two metrics used to estimate the impact the operational environment will have on the failure rate. These data items; the amount of system overhead, the amount of execution time, and the number of exception conditions encountered during an hour of operating time, can be derived from the test environment, estimated, or calculated from a benchmark. In the first case, the data can be collected from the test environment and, based on the assumption that the test environment is representative of the operational environment, used for the metric calculation. In the second case, sample data can be collected from the test environment and based on an experienced analyst's judgement, that data can be adjusted to represent the relative workload and stress differences expected between the test environment and operational environment. In the third case, a benchmark can be run in the operational environment to provide the data.

    The data required is typically available from mainframe vendor operating system utilities. It is more difficult to collect in an embedded computer application where the target computer may be a special processor without a significant operating system capability.

    To collect the data, monitor the processing during a specified time period (test period). This time period should be representative, or as close as possible, of the operational environment. During that time period collect the identified information and record it on Worksheet 18.

16. Potential/Plans for Automation: Information concerning workload and overhead is routinely gathered in automated computer management systems.

17. Remarks: It should be noted that this metric could also be the difference between idle time and total time. In a time-shared system, a significant portion of the busy time may be occupied by system overhead. In addition, in some applications, time that would otherwise be idle is absorbed by low-priority tasks (such as checking data bases for consistency) that would otherwise be idle. If a low-priority task is used to soak up idle time, it may produce a misleading estimate of the actually busy time -- i.e., the time used by higher-priority tasks.

WORKSHEET 18
OPERATING ENVIRONMENT DATA

Total Execution Time ET = __________

Amount of Operating System Overhead time: OS = __________

Number of exception conditions encountered: NEC = __________

Number of exception conditions encountered per hour of execution then is EC = NEC/ET = __________

APPENDIX C

OPTIONAL ANSWER SHEETS

METRIC ANSWER SHEET 1

SYSTEM LEVEL
PRE-SOFTWARE REQUIREMENTS PHASE

PROJECT________________________ ANALYST____________________

CODE SAMPLE ID____________________ PARENT ID____________________

DATA SOURCE(S)____

____

____

I. Metric Work Sheet 0/ Date________

Application Type__________ (from Task 101)

II. Metric Work Sheet 1A/ Date________

Development Environment__________ (from Task 102)

III. Metric Work Sheet 1B/ Date________

1a Y N
1b Y N
1c Y N
1d Y N
1e Y N
1f Y N
1g Y N UNK
1h Y N UNK

2a Y N
2b Y N
2c Y N
2d Y N
2e Y N
2f Y N
2g Y N
2h Y N
2i Y N

3a Y N
3b Y N
3c Y N
3d Y N
3e Y N
3f Y N
3g Y N
3h Y N
3i Y N
3j Y N
3k Y N

4a Y N
4b Y N
4c Y N
4d Y N
4e Y N
4f Y N
4g Y N
4h Y N
4i Y N
4j Y N

5a Y N
5b Y N
5c Y N
5d Y N
5e Y N
5f Y N

METRIC ANSWER SHEET 2A

CSCI LEVEL
SOFTWARE REQUIREMENTS ANALYSIS PHASE

PROJECT ____________________ ANALYST ____________________

CODE SAMPLE ID ____________________ PARENT ID ____________________

DATA SOURCE(S) ____

____

____

I. Metric Work Sheet 2A/ Date ________

AM.1(1) a ______

b ______

c ______

d Y N NA UNK

AM.2(1) a ______

b ______

c ______

d Y N NA UNK

AM.1(3) Y N NA UNK

AM.1(4) a ______

b ______

c ______

d Y N NA UNK

AM.2(1) Y N NA UNK

AM.3(1) Y N NA UNK

AM.3(2) Y N NA UNK

AM.3(3) Y N NA UNK

AM.3(4) Y N NA UNK

AM.4(1) Y N NA UNK

AM.5(1) Y N NA UNK

AM.6(1) Y N NA UNK

AM.7(1) Y N NA UNK

AM.7(2) Y N NA UNK

AM.7(3) Y N NA UNK

RE.1(1) Y N NA UNK

RE.1(2) Y N NA UNK

RE.1(3) Y N NA UNK

RE.1(4) Y N NA UNK

II. Metric Work Sheet 3A/ Date ________

TC.1(1) Y N

ST SCORE ______

III. Metric Work Sheet 10A/ Date ________

NR ______

AC.1(3) Y N NA UNK

AC.1(4) Y N NA UNK

AC.1(5) Y N NA UNK

AC.1(6) Y N NA UNK

AU.1(1) Y N NA UNK

AU.2(1) Y N NA UNK

AU.2(2) Y N NA UNK

CP.1(1) Y N NA UNK

CP.1(2) a ______

b ______

c ______

d Y N NA UNK

CP.1(3) a ______

b ______

c ______

d Y N NA UNK

CP.1(5) Y N NA UNK

CP.1(6) Y N NA UNK

CP.1(7) Y N NA UNK

CP.1(8) Y N NA UNK

CS.1(1) Y N NA UNK

CS.1(2) Y N NA UNK

CS.1(3) Y N NA UNK

CS.1(4) Y N NA UNK

CS.1(5) Y N NA UNK

CS.2(1) Y N NA UNK

CS.2(2) Y N NA UNK

CS.2(3) Y N NA UNK

CS.2(4) Y N NA UNK

CS.2(5) Y N NA UNK

CS.2(6) Y N NA UNK

METRIC ANSWER SHEET 2B

CSCI LEVEL
PRELIMINARY DESIGN PHASE

PROJECT________________________ ANALYST____________________

CODE SAMPLE ID____________________ PARENT ID____________________

DATA SOURCE(S)____

____

____

I. Metric Work Sheet 2B/ Date________

AM.3(1)	Y	N	NA	UNK
AM.4(1)	Y	N	NA	UNK
AM.5(1)	Y	N	NA	UNK
AM.6(1)	Y	N	NA	UNK
AM.6(2)	Y	N	NA	UNK
AM.6(3)	Y	N	NA	UNK
AM.6(4)	Y	N	NA	UNK
AM.7(1)	Y	N	NA	UNK
AM.7(2)	Y	N	NA	UNK
AM.7(3)	Y	N	NA	UNK
RE.1(1)	Y	N	NA	UNK
RE.1(2)	Y	N	NA	UNK
RE.1(3)	Y	N	NA	UNK
RE.1(4)	Y	N	NA	UNK

II. Metric Work Sheet 3B/ Date__________

TC.1(1)	Y	N

ST SCORE ______

III. Metric Work Sheet 10B/ Date________

AC.1(7)		Y	N	NA	UNK
AU.1(1)		Y	N	NA	UNK
AU.1(4)	a______				
	b______				
	c______				
	d	Y	N	NA	UNK
AU.2(2)		Y	N	NA	UNK
CP.1(1)		Y	N	NA	UNK
CP.1(2)	a______				
	b______				
	c______				
	d	Y	N	NA	UNK
CP.1(3)	a______				
	b______				
	c______				
	d	Y	N	NA	UNK
CP.1(4)	a______				
	b______				
	c______				
	d	Y	N	NA	UNK
CP.1(6)		Y	N	NA	UNK
CP.1(9)		Y	N	NA	UNK
CP.1(11)		Y	N	NA	UNK
CS.1(1)		Y	N	NA	UNK
CS.1(5)		Y	N	NA	UNK
CS.2(1)		Y	N	NA	UNK
CS.2(2)		Y	N	NA	UNK
CS.2(3)		Y	N	NA	UNK
CS.2(4)		Y	N	NA	UNK
CS.2(5)		Y	N	NA	UNK
CS.2(6)		Y	N	NA	UNK

METRIC ANSWER SHEET 2C

UNIT LEVEL
DETAILED DESIGN PHASE

PROJECT______________________ ANALYST______________________

CODE SAMPLE ID______________________ PARENT ID______________________

DATA SOURCE(S)______________________

____

____

I. Metric Work Sheet 2C/ Date__________

AM.1(3)	Y	N	NA	UNK
AM.2(7)	Y	N	NA	UNK

II. Metric Work Sheet 10C/ Date__________

CP.1(1)	Y	N	NA	UNK
CP.1(2)	a_______			
	b_______			
CP.1(4)	a_______			
	b_______			
CP.1(9)	Y	N	NA	UNK
CP.1(10)	Y	N	NA	UNK
CS.1(1)	Y	N	NA	UNK
CS.1(2)	Y	N	NA	UNK
CS.1(3)	Y	N	NA	UNK
CS.1(4)	Y	N	NA	UNK
CS.1(5)	Y	N	NA	UNK
CS.2(1)	Y	N	NA	UNK
CS.2(2)	Y	N	NA	UNK
CS.2(3)	Y	N	NA	UNK
CS.2(6)	Y	N	NA	UNK

METRIC ANSWER SHEET 2D

CSCI LEVEL
DETAILED DESIGN PHASE

PROJECT ____________________ ANALYST ____________________

CODE SAMPLE ID ____________________ PARENT ID ____________________

DATA SOURCE(S) ____________________

____

____

I. Metric Work Sheet 2D/ Date ________

AM.1(3) a ______
b ______
c ______
d Y N NA UNK

AM.2(2) Y N NA UNK
AM.2(3) Y N NA UNK
AM.2(4) Y N NA UNK
AM.2(5) Y N NA UNK
AM.2(6) Y N NA UNK

AM.2(7) a ______
b ______
c ______
d Y N NA UNK

AM.3(2) Y N NA UNK
AM.3(3) Y N NA UNK
AM.3(4) Y N NA UNK

II. Metric Work Sheet 3C/ Date ________

TC.1(1) Y N
TC.1(2) Y N
ST SCORE ______

III. Metric Work Sheet 10D/ Date ________

AU.1(2) a ______
b ______
c ______
d Y N NA UNK

AU.1(3) a ______
b ______
c ______
d Y N NA UNK

Metric Work Sheet 10D (Cont.)

AU.1(4) a ______
b ______
c ______
d Y N NA UNK

CP.1(1) a ______
b ______
c Y N NA UNK

CP.1(2) a ______
b ______
c ______
d Y N NA UNK

CP.1(3) a ______
b ______
c ______
d Y N NA UNK

CP.1(4) a ______
b ______
c ______
d Y N NA UNK

CP.1(9) a ______
b ______
c Y N NA UNK

CP.1(10) a ______
b ______
c Y N NA UNK

CS.1(1) a ______
b ______
c Y N NA UNK

METRIC ANSWER SHEET 2D (cont.)

CSCI LEVEL
DETAILED DESIGN PHASE

PROJECT________________________ ANALYST________________________

CODE SAMPLE ID________________________ PARENT ID________________________

DATA SOURCE(S)____

____

____

Metric Work Sheet 10D (Cont.)

CS.1(2) a______

b______

c Y N NA UNK

CS.1(3) a______

b______

c Y N NA UNK

CS.1(4) a______

b______

c Y N NA UNK

CS.1(5) a______

b______

c Y N NA UNK

CS.2(1) a______

b______

c Y N NA UNK

CS.2(2) a______

b______

c Y N NA UNK

CS.2(3) a______

b______

c Y N NA UNK

CS.2(6) a______

b______

c Y N NA UNK

METRIC ANSWER SHEET 3
CSCI LEVEL
PRELIMINARY DESIGN PHASE

PROJECT________________________ ANALYST________________________

CODE SAMPLE ID________________________ PARENT ID________________________

DATA SOURCE(S)____

____

____

I. Metric Work Sheet 2B/ Date________

AM.3(1)	Y	N	NA	UNK
AM.4(1)	Y	N	NA	UNK
AM.5(1)	Y	N	NA	UNK
AM.6(1)	Y	N	NA	UNK
AM.6(2)	Y	N	NA	UNK
AM.6(3)	Y	N	NA	UNK
AM.6(4)	Y	N	NA	UNK
AM.7(1)	Y	N	NA	UNK
AM.7(2)	Y	N	NA	UNK
AM.7(3)	Y	N	NA	UNK
RE.1(1)	Y	N	NA	UNK
RE.1(2)	Y	N	NA	UNK
RE.1(3)	Y	N	NA	UNK
RE.1(4)	Y	N	NA	UNK

II. Metric Work Sheet 3B/ Date________

TC.1(1)	Y	N

ST SCORE________

III. Metric Work Sheet 10B/ Date________

AC.1(7)		Y	N	NA	UNK
AU.1(1)		Y	N	NA	UNK
AU.1(4)	a________				
	b________				
	c________				
	d	Y	N	NA	UNK
AU.2(2)		Y	N	NA	UNK
CP.1(1)		Y	N	NA	UNK
CP.1(2)	a________				
	b________				
	c________				
	d	Y	N	NA	UNK
CP.1(3)	a________				
	b________				
	c________				
	d	Y	N	NA	UNK
CP.1(4)	a________				
	b________				
	c________				
	d	Y	N	NA	UNK
CP.1(6)		Y	N	NA	UNK
CP.1(9)		Y	N	NA	UNK
CP.1(11)		Y	N	NA	UNK
CS.1(1)		Y	N	NA	UNK
CS.1(5)		Y	N	NA	UNK
CS.2(1)		Y	N	NA	UNK
CS.2(2)		Y	N	NA	UNK
CS.2(3)		Y	N	NA	UNK
CS.2(4)		Y	N	NA	UNK
CS.2(5)		Y	N	NA	UNK
CS.2(6)		Y	N	NA	UNK

METRIC ANSWER SHEET 4
UNIT LEVEL
DETAILED DESIGN PHASE

PROJECT__________ ANALYST__________

CODE SAMPLE ID__________ PARENT ID__________

DATA SOURCE(S)__________

____

____

I. Metric Work Sheet 2C/ Date______

AM.1(3)	Y	N	NA	UNK
AM.2(7)	Y	N	NA	UNK

II. Metric Work Sheet 10C/ Date______

CP.1(1)	Y	N	NA	UNK
CP.1(2)	a______			
	b______			
CP.1(4)	a______			
	b______			
CP.1(9)	Y	N	NA	UNK
CP.1(10)	Y	N	NA	UNK
CS.1(1)	Y	N	NA	UNK
CS.1(2)	Y	N	NA	UNK
CS.1(3)	Y	N	NA	UNK
CS.1(4)	Y	N	NA	UNK
CS.1(5)	Y	N	NA	UNK
CS.2(1)	Y	N	NA	UNK
CS.2(2)	Y	N	NA	UNK
CS.2(3)	Y	N	NA	UNK
CS.2(6)	Y	N	NA	UNK

METRIC ANSWER SHEET 5
CSCI LEVEL
DETAILED DESIGN PHASE

PROJECT______________________ ANALYST______________________

CODE SAMPLE ID______________________ PARENT ID______________________

DATA SOURCE(S)____

____

____

I. Metric Work Sheet 2D/ Date________

AM.1(3) a______
b______
c______
d Y N NA UNK

AM.2(2) Y N NA UNK

AM.2(3) Y N NA UNK

AM.2(4) Y N NA UNK

AM.2(5) Y N NA UNK

AM.2(6) Y N NA UNK

AM.2(7) a______
b______
c______
d Y N NA UNK

AM.3(2) Y N NA UNK

AM.3(3) Y N NA UNK

AM.3(4) Y N NA UNK

II. Metric Work Sheet 3C/ Date________

TC.1(1) Y N

TC.1(2) Y N

ST SCORE ______

III. Metric Work Sheet 10D/ Date________

AU.1(2) a______
b______
c______
d Y N NA UNK

AU.1(3) a______
b______
c______
d Y N NA UNK

Metric Work Sheet 10D (Cont.)

AU.1(4) a______
b______
c______
d Y N NA UNK

CP.1(1) a______
b______
c Y N NA UNK

CP.1(2) a______
b______
c______
d Y N NA UNK

CP.1(3) a______
b______
c______
d Y N NA UNK

CP.1(4) a______
b______
c______
d Y N NA UNK

CP.1(9) a______
b______
c Y N NA UNK

CP.1(10) a______
b______
c Y N NA UNK

CP.1(11) Y N NA UNK

CS.1(1) a______
b______
c Y N NA UNK

METRIC ANSWER SHEET 5 (cont.)

CSCI LEVEL
DETAILED DESIGN PHASE

PROJECT__________________________ ANALYST______________________

CODE SAMPLE ID_____________________ PARENT ID__________________

DATA SOURCE(S)____

____

____

Metric Work Sheet 10D (Cont.)

CS.1(2) a______
b______
c Y N NA UNK

CS.1(3) a______
b______
c Y N NA UNK

CS.1(4) a______
b______
c Y N NA UNK

CS.1(5) a______
b______
c Y N NA UNK

CS.2(1) a______
b______
c Y N NA UNK

CS.2(2) a______
b______
c Y N NA UNK

CS.2(3) a______
b______
c Y N NA UNK

CS.2(6) a______
b______
c Y N NA UNK

METRIC ANSWER SHEET 6

UNIT LEVEL
CODING AND UNIT TESTING
PHASE

PROJECT____________________ ANALYST____________________

CODE SAMPLE ID____________________ PARENT ID____________________

DATA SOURCE(S)____

____

____

I. Metric Work Sheet 4A/ Date__________

1.a. MLOC = __________

b. ALOC = __________

c. HLOC = __________

2. sx = __________

II. Metric Work Sheet 11A/ Date__________

MO.1(3) Y N NA UNK

MO.1(4) a______
b______
c______

MO.1(5) Y N NA UNK

MO.1(6) Y N NA UNK

MO.1(7) Y N NA UNK

MO.1(8) Y N NA UNK

MO.1(9) Y N NA UNK

MO.2(5) Y N NA UNK

SI.1(2) Y N NA UNK

SI.1(3) Y N NA UNK

SI.1(4) Y N NA UNK

SI.1(5) a______
b______
c______
d Y N NA UNK

SI.1(10) Y N NA UNK

SI.4(1) Y N NA UNK

SI.4(2) a______ b______

SI.4(3) a______
b______
c______

II. Metric Work Sheet 11A (Cont.)

SI.4(4) a______
b______
c______

SI.4(5) Y N NA UNK

SI.4(6) a______
b______

SI.4(7) a______
b______

SI.4(8) a______
b______

SI.4(9) a______
b______
c______

SI.4(10) a______
b______
c______

SI.4(11) ______

SI.4(12) Y N NA UNK

SI.4(13) Y N NA UNK

SI.5(1) a______
b______

SI.5(2) a______
b______
c______

SI.5(3) Y N NA UNK

METRIC ANSWER SHEET 7

CSCI LEVEL
CODING AND UNIT TESTING

PROJECT__________________ ANALYST__________________
CODE SAMPLE ID__________________ PARENT ID__________________
DATA SOURCE(S) __________________

I. Metric Work Sheet
4B/Date________
1. a______
b______
c______
2. a______
b______
c______
d______
3. u______
w______
x______

II. Metric Work Sheet
11B/Date________

MO.1(2) Y N NA UNK
MO.1(3) a______
b______
c______
d Y N NA UNK
MO.1(4) a______
b______
c______
MO.1(5) a______
b______
c Y N NA UNK
MO.1(6) a______
b______
c Y N NA UNK
MO.1(7) a______
b______
c Y N NA UNK

MO.1(8) a______
b______
c Y N NA UNK
MO.1(9) a______
b______
c Y N NA UNK
SI.1(2) a______
b______
c Y N NA UNK
SI.1(3) a______
b______
c Y N NA UNK
SI.1(4) a______
b______
c Y N NA UNK
SI.1(5) a______
b______
c Y N NA UNK
SI.1(10) Y N NA UNK
SI.2(1) a______
b______
c Y N NA UNK
SI.4(1) a______
b______
c Y N NA UNK
SI.4(2) a______
b______
c
SI.4(3) a______
b______
c

SI.4(4) a______
b______
c______
SI.4(5) a______
b______
c Y N NA UNK

METRIC ANSWER SHEET 7 (cont.)

CSCI LEVEL
CODING AND UNIT TESTING

PROJECT__________ ANALYST__________

CODE SAMPLE ID__________ PARENT ID__________

DATA SOURCE(S) __________

II. Metric Work Sheet 11B/Date______

SI.4(6) a_____
b_____

SI.4(7) a_____
b_____

SI.4(8) a_____
b_____

SI.4(9) a_____
b_____
c_____

SI.4(10) a_____
b_____
c_____

SI.4(11) a_____

SI.4(12) a_____
b_____
c Y N NA UNK

SI.4(13) a_____
b_____
c Y N NA UNK

SI.4(14) Y N NA UNK

SI.5(1) a_____
b_____

SI.5(2) a_____
b_____
c_____

SI.5(3) a_____
b_____
c Y N NA UNK

METRIC ANSWER SHEET 8
CSCI LEVEL
CODING AND UNIT TESTING

PROJECT____________________ ANALYST____________________

CODE SAMPLE ID____________________ PARENT ID____________________

DATA SOURCE(S) ____

____

____

I. Metric Work Sheet
4B/Date________

1. a________
b________
c________
2. a________
b________
c________
d________
3. u________
w________
x________

II. Metric Work Sheet
11B/Date________

MO.1(2) Y N NA UNK
MO.1(3) a________
b________
c________
d Y N NA UNK
MO.1(4) a________
b________
c________
d________
MO.1(5) a________
b________
c Y N NA UNK
MO.1(6) a________
b________
c Y N NA UNK
MO.1(7) a________ NA UNK
b________ NA UNK
c Y N NA UNK

MO.1(8) a________
b________
c Y N NA UNK
MO.1(9) a________
b________
c Y N NA UNK
SI.1(2) a________
b________
c Y N NA UNK
SI.1(3) a________
b________
c Y N NA UNK
SI.1(4) a________
b________
c Y N NA UNK
SI.1(5) a________
b________
c Y N NA UNK
SI.1(10) Y N NA UNK
SI.2(1) a________
b________
c Y N NA UNK
SI.4(1) a________
b________
c Y N NA UNK
SI.4(2) a________
b________
c________
SI.4(3) a________
b________
c________

SI.4(4) a________
b________
c________
SI.4(5) a________
b________
c Y N NA UNK

METRIC ANSWER SHEET 8 (Con't.)
CSCI LEVEL
CODING AND UNIT TESTING

PROJECT____________________ ANALYST____________________

CODE SAMPLE ID____________________ PARENT ID____________________

DATA SOURCE(S) ____

____

____

II. Metric Work Sheet 11B/Date________

SI.4(6) a______
b______

SI.4(7) a______
b______

SI.4(8) a______
b______

SI.4(9) a______
b______
c______

SI.4(10) a______
b______
c______

SI.4(11) a______

SI.4(12) a______
b______
c Y N NA UNK

SI.4(13) a______
b______
c Y N NA UNK

SI.4(14) Y N NA UNK

SI.5(1) a______
b______

SI.5(2) a______
b______
c______

SI.5(3) a______
b______
c Y N NA UNK

APPENDIX D

SOFTWARE TESTING TECHNIQUES & TOOLS

APPENDIX D

SOFTWARE TESTING TECHNIQUES & TOOLS

D.1 Preparing and Executing the Tests

This appendix desicribes the overall testing process for the six techniques identified in Task Section 200. Detailed instructions for the application of each testing technique are provided. A set of useful test/support tools also are described.

Each testing technique creates test cases, test procedures and test drivers from specifications. A test harness can be setup to provide inputs to the test driver and to capture and compare test outputs from the code samples under test.

Figure D-1 illustrates the overall flow of test case preparation and execution for each of the testing techniques. When two or more techniques are used together at the same test level their tests should be prepared, executed and analyzed together.

D.1.1 Test Preparation

Preparing for test execution includes test data preparation and formulation of expected results. Test data preparation formulates test cases and the data to be input to the program. Test case preparation is not applicable to the static testing techniques. For the dynamic testing techniques, test preparation is accomplished through both manual and automated methods.

Test cases are chosen as a result of analyzing the requirements and design specifications and the code itself. Test data is prepared to demonstrate and exercise externally visible functions, program structures, data structures, and internal functions. Each test case includes a set of input data and the expected results. The expected results are expressed in terms of final values or intermediate states of program execution.

Testers develop test cases, test data and expected results through examining the program specifications (in particular, the design and program code) according to the procedure of a particular testing technique. Test cases have the objective of demonstrating that the functions, interface requirements, and solution constraints are satisfied according to the test objectives. Test cases are determined from the inputs, functions and structures of the design and code. Test data is determined from the program to exercise computational structures implemented within the program code.

The final step before test execution is to tailor the test driver (if needed) to suit any particular needs of the test cases once they have been developed.

D.1.2 Test Execution

For the dynamic techniques, test execution involves executing a program with prepared test cases and then collecting the results. For the static techniques, test execution involves the execution of an automated test tool and evaluation of the applicable hardcopy outputs.

Dynamic testing can be performed in a bottom-up fashion. Bottom-up testing consists of testing individual units and low level CSCs. This requires the use of test drivers and stubs or dummy routines for interfacing units and CSC not under test.

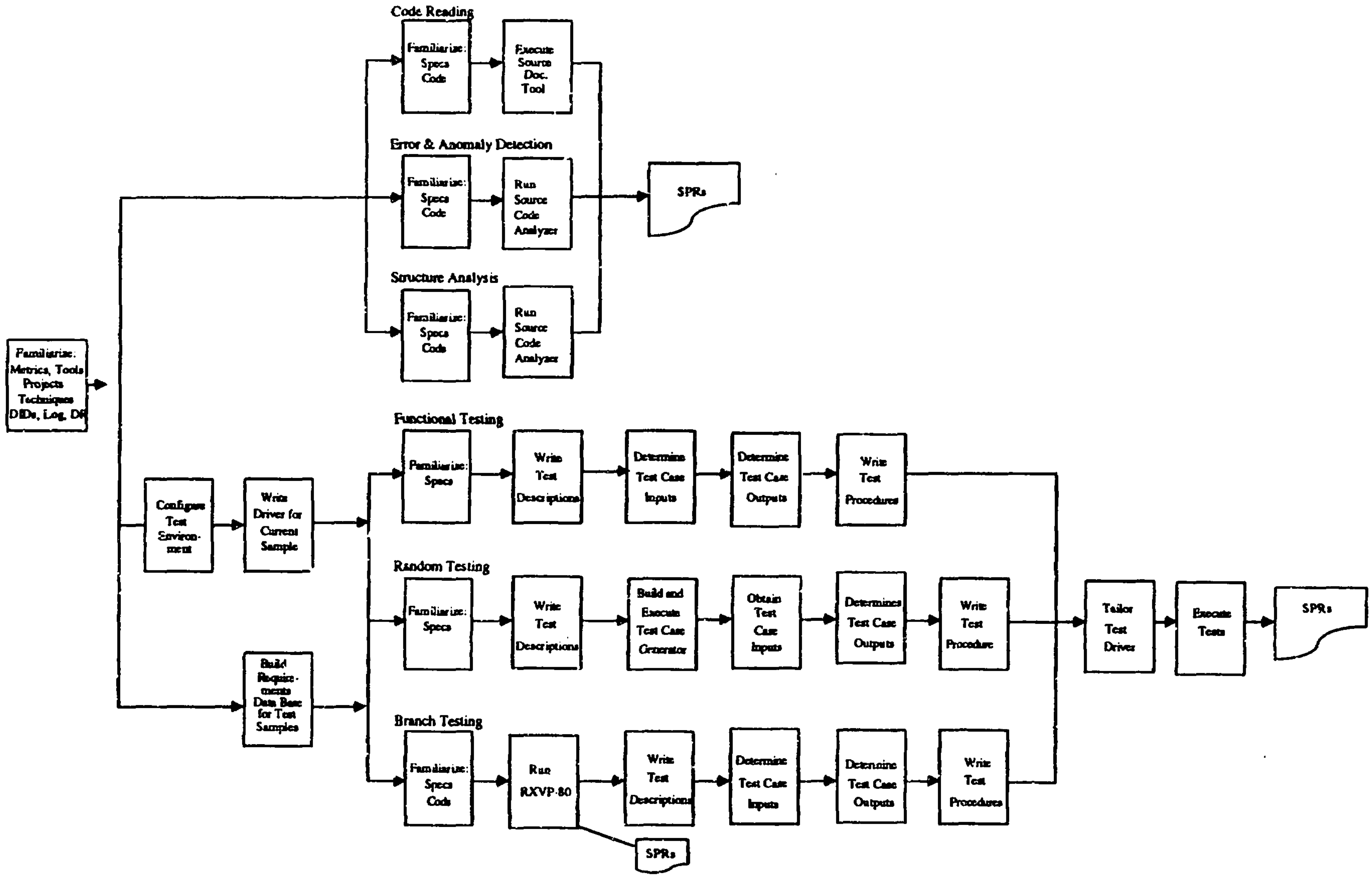


Figure D-1 Preparing and Executing the Tests

D.1.3 Test Analysis

Test analysis is performed by the testers for the dynamic testing techniques to capture and report execution details (e.g., branch execution counts) and to determine the thoroughness of the testing. For the static testing techniques, analyses is an integral part of their execution.

The process of analyzing the-dynamic test results includes comparing the actual to expected outputs. This analysis requires a specification of the expected outputs for each test case. When the output data for non-interactive tests is machine readable, an automated comparator can be used. Interactive outputs are evaluated visually while the tests run. Upon completion of the test outputs analysis, further analyses of test coverage can be made for all execution paths, inputs, units, interfaces and requirements, as applicable to the test level.

D.2 Instructions for the Static Analysis Testing Techniques

Static analysis detects errors by examining the software system (i.e., requirements statement, program code, users manual) rather than by executing it. Some examples of the errors detected are language syntax errors, misspellings, incorrect punctuation, improper sequencing of statements, and missing specification elements. Static analysis techniques may be manually or automatically applied, although automated techniques require a machine-readable specification of the product.

Manual static analysis techniques may be applied to all development products such as the requirements statements, program code, or a users manual. In general, these techniques are straightforward and when applied with discipline are effective in preventing and detecting errors.

D.2.1 Code Review

This static testing technique involves the visual inspection of a program with the objective of finding faults. Test personnel perform an inspection of the sample code units using the appropriate documentation and specifications. These code inspections are driven by checklists in order to identify common types of errors in the code. Appendix C contains the checklists which were used in the FORTRAN-based studies described in Volume 1 of this report. Figure D-2 illustrates the process.

Process

Initially, the source code can be processed statically by a documentation tool such as SDDL. Thus the source is enhanced with indentation based on structure logic and with flow line arrows in a two-dimensional representation. Other outputs of SDDL for organizational use in the code inspection are the table of contents, a module cross reference listing, and a module reference tree. The tester works through the checklist sequentially, referring to the annotated source listing and above mentioned organizational aids.

The tester looks for errors (and often for poor programming practices) in the source code and comments by examining it for attributes noted in the checklist. This checklist identifies all aspects of the code to be studied for problems and all checks to be made for agreement between the code and the specifications. Examples of code attributes in the checklist are: 1) whether branch points and loop indices have been coded correctly; 2) whether formal and actual parameters are consistent and correct; and 3) whether specifications, inline comments, and code are consistent and code is complete with respect to the specifications. When a problem or error is found during the code review, an SPR is completed.

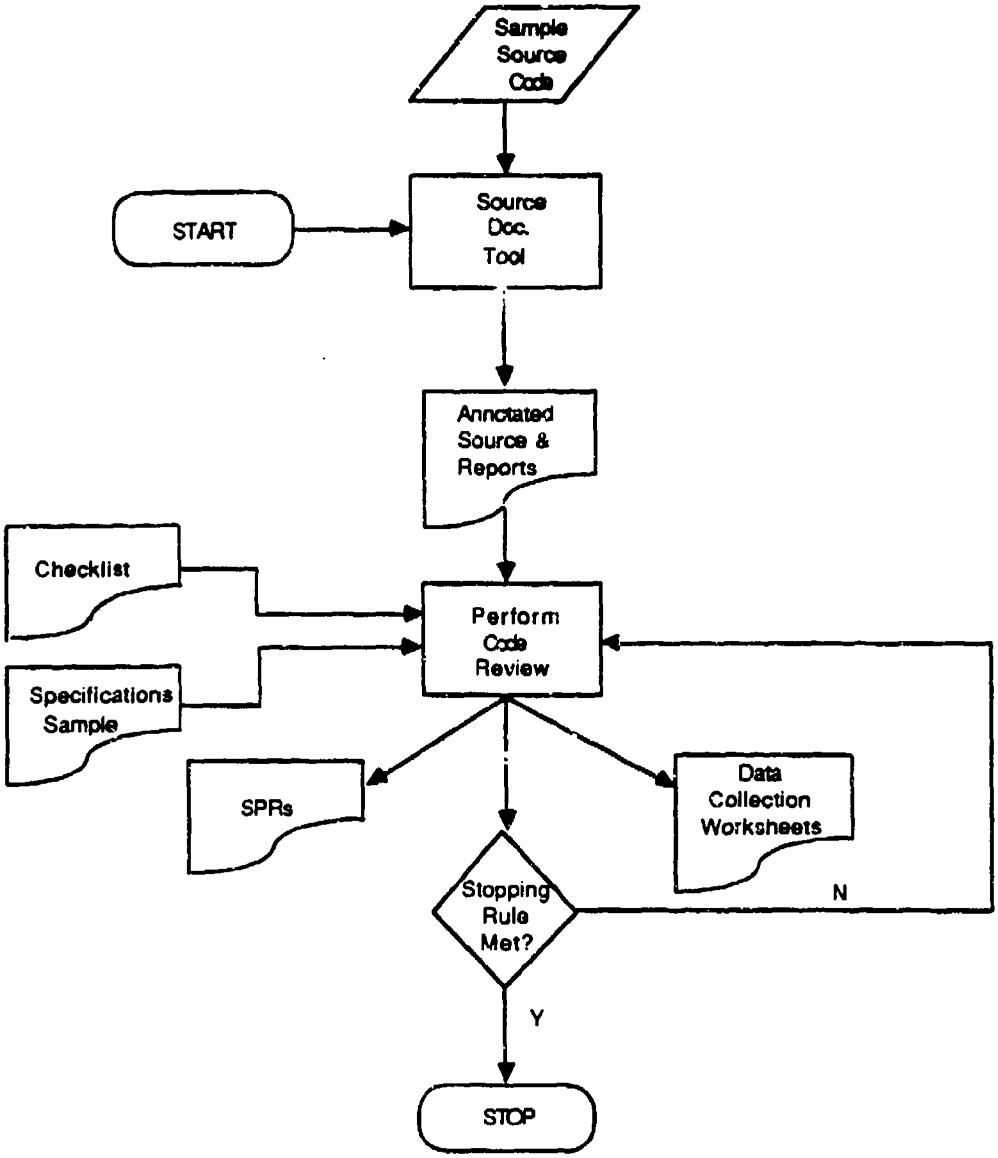


Figure D-2 Code Review Activity Flow

Information Input

Inputs to code review are 1) the sample source code, 2) the code review checklist, and 3) the relevant specifications.

Information Output

Outputs from code review are: 1) an SPR for each error found, and 2) test activity worksheets.

D.2.2 Error & Anomaly Detection

This static testing technique is applied to detect faults via syntax checking, data checking and physical units checking of the source code. Syntax checking includes checks for uninitialized variables and unreachable code. Data checking entails identifying set-use anomalies, conducting a data flow analysis and unit consistency checking. Physical unit checking locks for consistency in physical units usage. Test personnel perform error and anomaly detection on the selected sample code units using the appropriate documentation and specifications. Figure D-3 illustrates the process.

Process

Error and anomaly detection is applied using the following static analysis functions of an automated test tool such as RXVP-80:

a. Syntax checking: uninitialized variable screening, and unreachable code screening.

b. Data checking: data flow set/use anomalies, interface completeness and consistency (CSC integration level).

c. Physical Units Checking: checking for consistency in physical units (e.g. feet, gallons, liters, etc.) usage.

An SPR is completed for each error found; some errors may be reported directly in the test tool reports, some may be found by visual inspection of the code, and some by other means during the error and anomaly testing activities.

Information Input

Inputs to error and anomaly detection are: 1) the source code to be analyzed, 2) the specifications, and 3) the test tool.

Information Output

Outputs from error and anomaly detection are: 1) an SPR for every error found, and 2) test activity worksheets.

D.2.3 Structure Analysis

This static testing technique detects faults concerning the control structures and code structures of the source code, and improper subprogram usage. Test personnel perform structure analysis on the source code using the appropriate documents and specifications. Structure analysis is performed to determine the presence of improper or incomplete control structures, structurally dead code, possible recursion in routine calls, routines which are never called, and attempts to call non-existent routines. Figure D-4 illustrates the process.

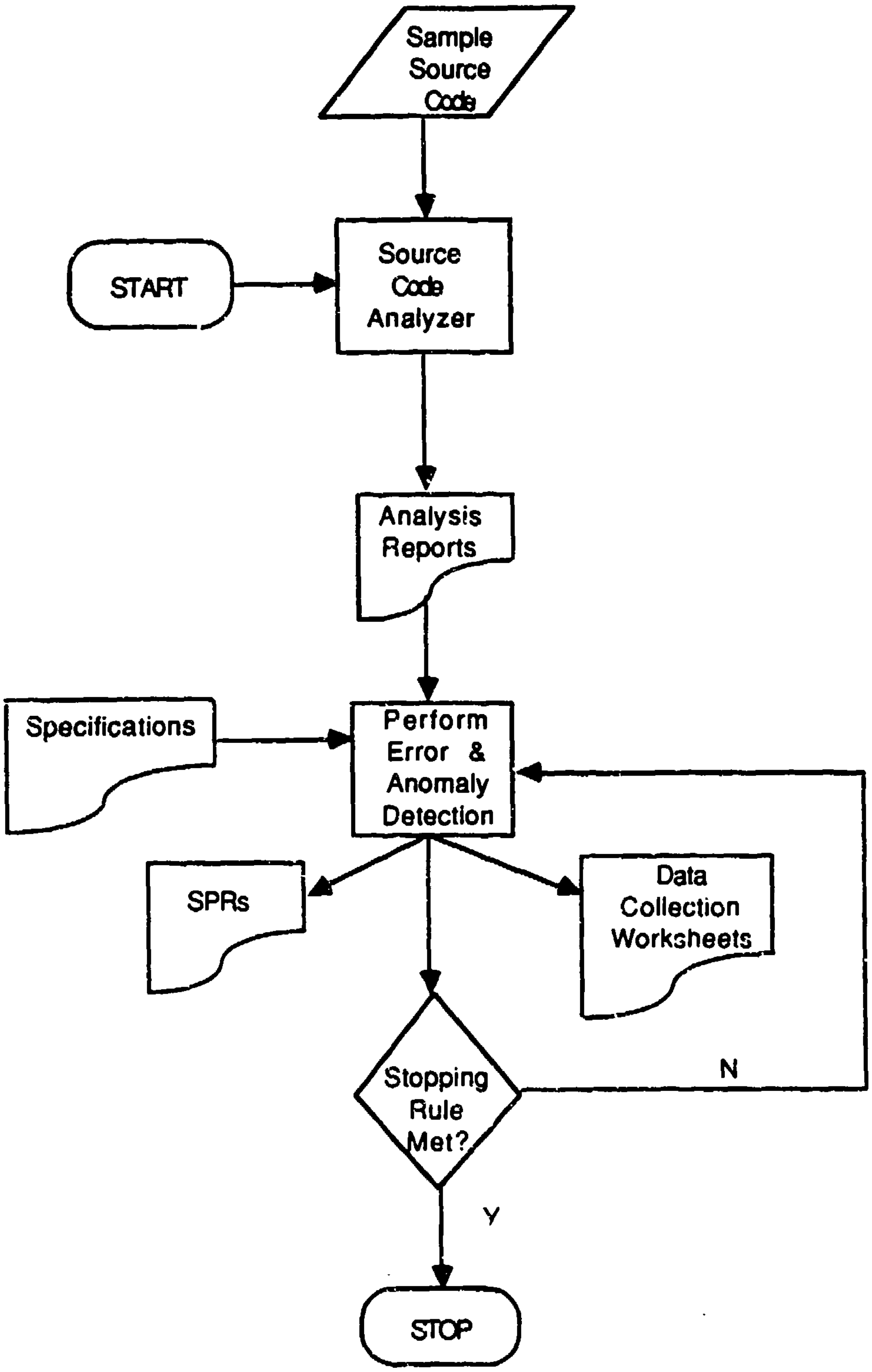


Figure D-3 Error and Anomaly Detection Activity Flow

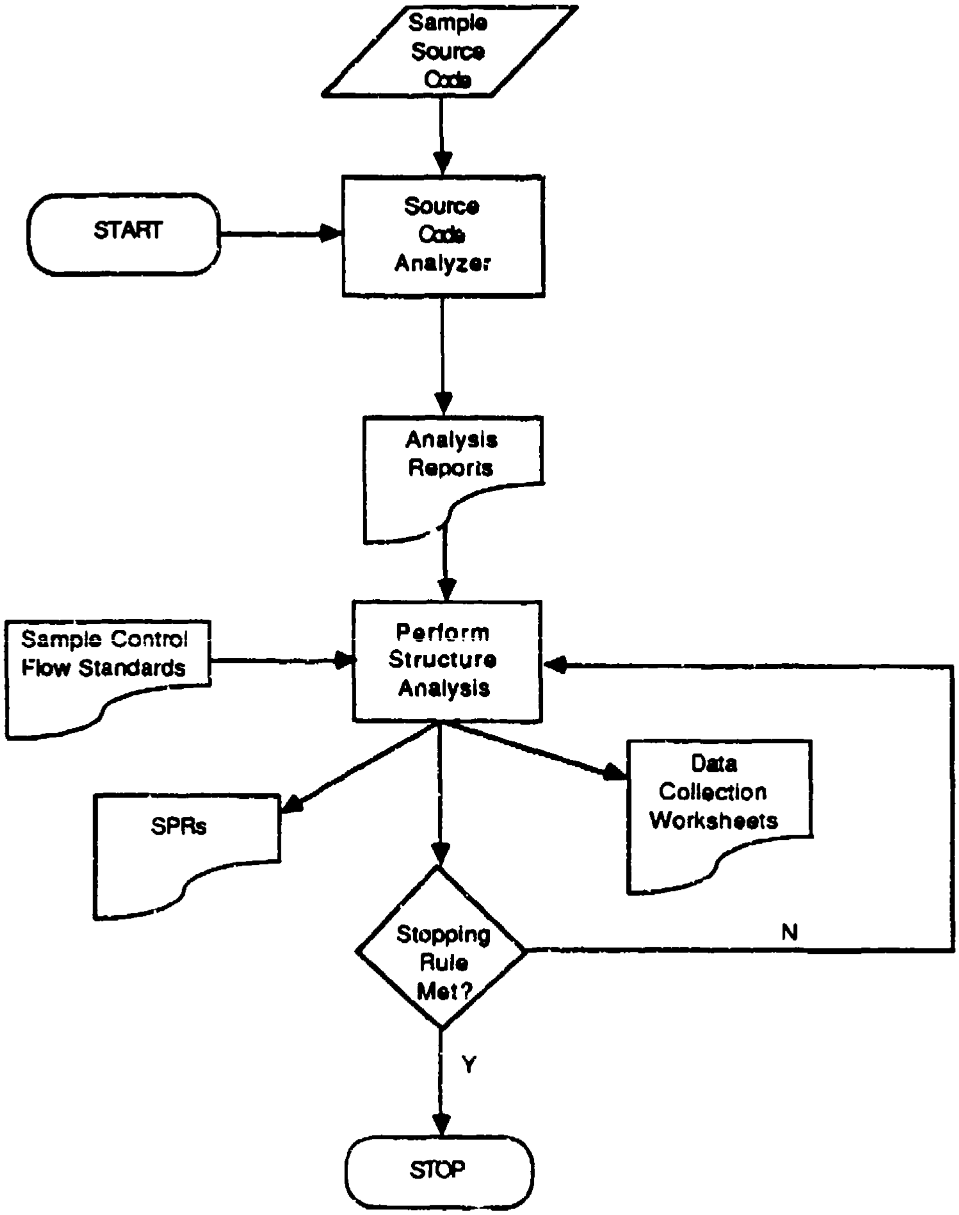


Figure D-4 Structure Analysis Activity Flow

Process

An automated tool such as RXVP-80 can be used to partially automate structure analysis. The test tool processes the source code and parameters describing control flow standards, and provides error reports and a program call graph. For example: analysis of graph cycles may indicate unintentionally recursive code; presence of a disjoint subset of the graph illustrates unreachable, or dead, code; and calls to nonexistent routines will be illustrated by edges with no sink nodes. The tester logs, the test tool error reports and errors illustrated by the call graph in SPRs. Any errors found by other means also are logged in SPRs.

Information Input

Inputs to structure analysis are: 1) the source code; 2) a specification of the control flow standards to be enforced in the language; and 3) the test tool.

Information Output

Outputs from structure analysis are: 1) an SPR for every error found, 2) a program call graph or report, and 3) test activity worksheets.

D.3 Instructions for the Dynamic Analysis Testing Techniques

In contrast to static analysis, which does not involve the execution of the program, dynamic analysis involves actual execution of the program. The principal applications of dynamic analysis includes software testing, debugging, and performance measurement. The three dynamic techniques that follow are best suited to software testing. Additional techniques for debugging and performance measurement are described in RADC TR 84-53. In general, dynamic analysis involves:

a. Preparing for test execution.
b. Test execution.
c. Analysis of test results.

Note that all dynamic techniques require an abstract specification of the program function as an input. According to DoD-STD-2167A, the specifications applicable at the unit and CSC levels are:

a. CSC level: Software Top Level Design Document (STLDD) or equivalent

b. Unit level: the Detailed Design Document (DDD)or equivalent.

D.3.1 Branch Testing

This testing technique combines static and dynamic techniques and detects failures. The static portion is used to determine test data that force selected branches to be executed. The dynamic portion is used to actually run the code with these test data and then obtain test outputs. Test personnel perform branch testing of the code samples using the appropriate documentation and specifications. Test coverage analysis is used to detect untested parts of the program. Output data analysis is applied to detect outputs that deviate from known or specified output values. Figure D-5 illustrates the process.

Process

Branch testing involves creating test cases that cause execution of a specified percentage of all branches in the code. Branch testing identifies all branches in the code under test. The tester uses

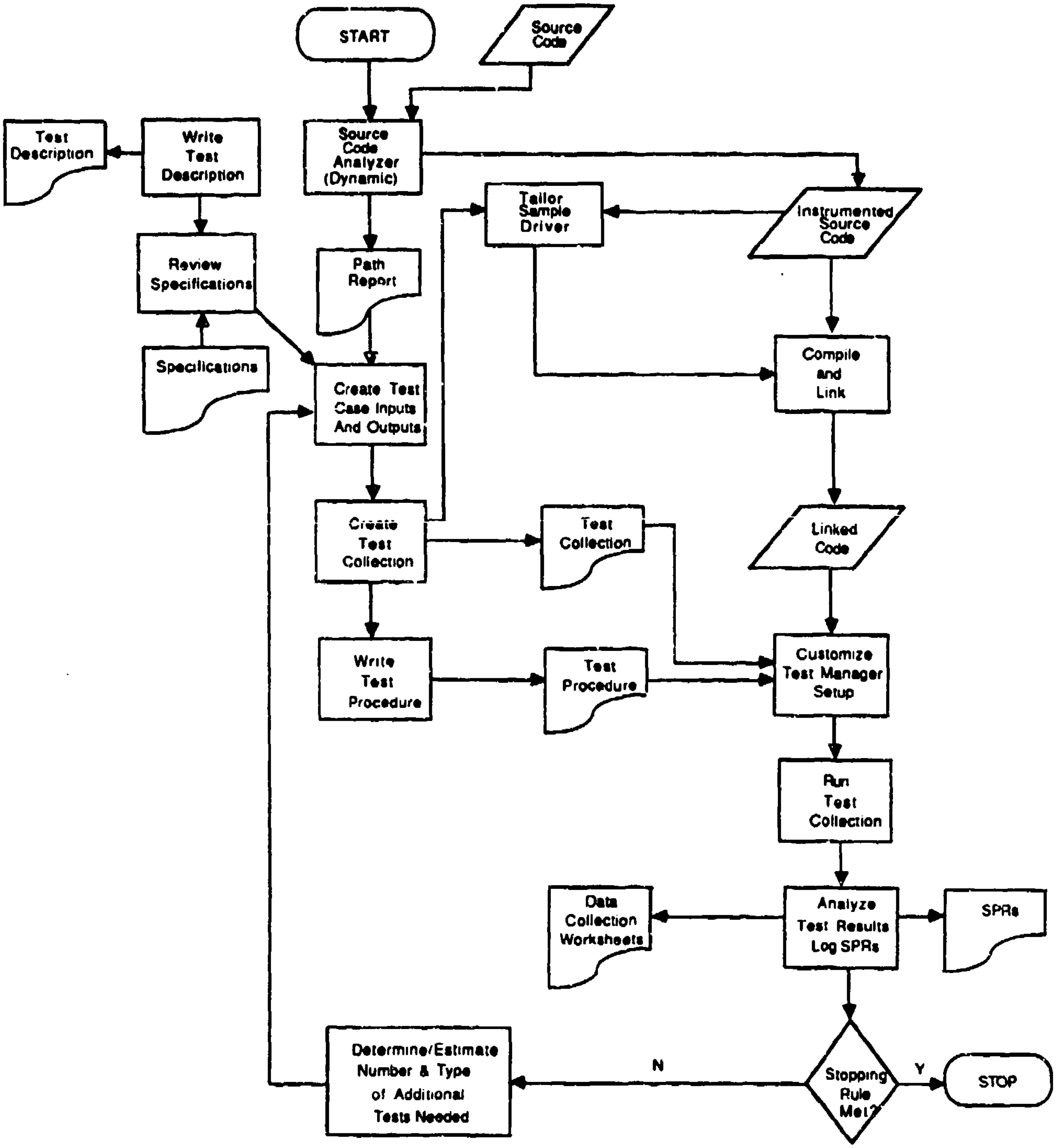


Figure D-5 Branch Testing Activity Flow

this information to create tests that will cause execution of each branch. A test tool such as RXVP-80 is used to identify all of the branches in the code. The tester determines the correct outputs expected from executing the code for each test case based on the specifications. Then, the tester writes a test procedure for these test cases, tailors the test driver as needed, and sets up the test collections and descriptions with a support tool such as the DEC Test Manager (DTM).

The next step is to execute the code under test with the tests created, and to track the branch coverage for each test case and for the total of all test cases. The test tool can be used to instrument the code under test. The tester executes the code with the given test case inputs by running the test collection, and uses the support tool to find differences in actual ana expected outputs. Execution of the sample code instrumented by the test tool generates reports on branch coverage. Testers review these reports to ensure that their test cases meet the stopping rule; if they don't, the tester returns to the static activity of creating more test cases to execute all branches at least twice. Testers compare expected outputs with outputs obtained by dynamic testing, and log all errors found on SPRs.

Information Input

Inputs to the static portion of branch testing are: 1) the source code of the sample and 2) the specifications of interest for the sample, and (3) the test tool.

Inputs to the dynamic portion are: 1) the source code of the sample, 2) the test cases and test procedures generated during the static portion of branch testing, and 3) the test tool and support tool.

Information Output

Outputs from the static portion of branch testing are: 1) test cases, 2) test procedures, and 3) test activity worksheets.

Outputs from the dynamic portion are: 1) the actual outputs, from executing the sample with each test case, 2) SPRs which document errors discovered in these outputs, 3) test case branch coverage reports, and 4) test activity worksheets.

D.3.2 Functional Testing

This dynamic testing technique finds failures consisting of discrepancies between the program and its specification. In using this testing technique, the program design is viewed as an abstract description of the design and requirement specifications. Test data are generated, based on knowledge of the programs under test and on the nature of the program's inputs. The test data are designed to ensure adequate testing of the requirements stated in the specification. Test personnel perform functional testing of the code sample using the appropriate documentation and specifications. Functional testing is performed to assure that each unit correctly performs the functions that it is intended to perform. Figure D-6 illustrates the process.

Process

Functional testing entails creating test cases to exercise all functions, or a given percentage of all functions, that the software specifications include as functional requirements. The tester consults the appropriate functional specifications provided, and manually creates test cases and corresponding test procedures to test all applicable functions. The test driver is tailored as needed, and a test support tool such as DTM is set up to run the tests as a test collection. The test sample is instrumented with a test tool such as RXVP-80 in order to gather path coverage information; note that this is not integral to functional testing, but is done to provide path coverage data of functional

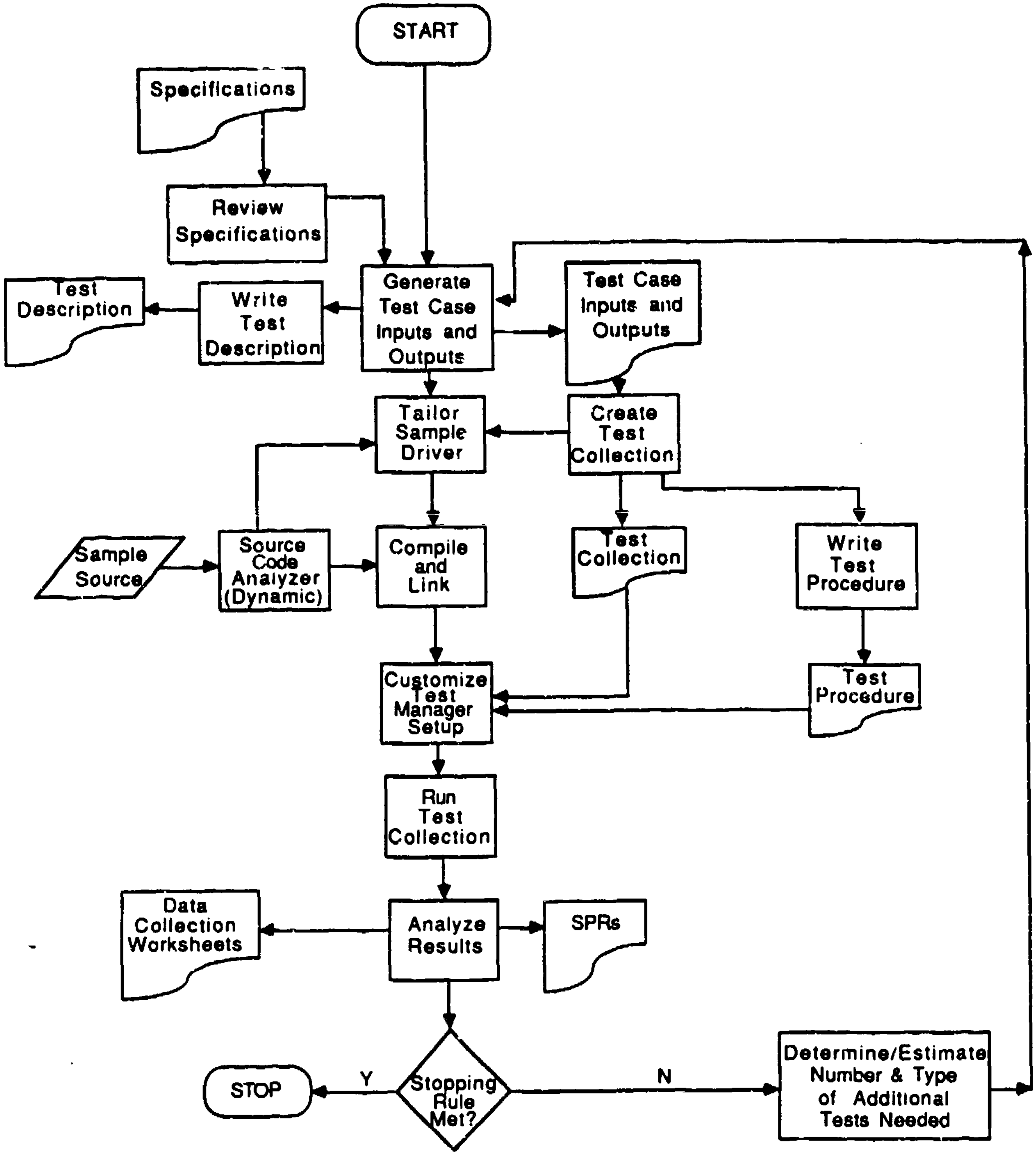


Figure D-6 Functional Testing Activity Flow

tests for experimental analyses. The tester executes the code with the given test case inputs by running the test collection, and uses the test support tool to find differences in actual and expected outputs. SPRs are logged for all errors found.

Information Input

Inputs to functional testing are: 1) an abstract specification of the program function, 2) the source code and 3) the test tool and support tool.

Information Output

Outputs of functional testing are: 1) test cases, 2) actual test outputs, 3) an SPR for each error found, and 4) test activity worksheets.

D.3.3 Random Testing

This dynamic testing technique tests a unit by randomly selecting subsets of all possible input values. The input subsets can be sampled according to the actual probability distribution of the input sequences, thus characterizing the usage; or according to other probability distribution. Random testing invokes bizarre combinations of input values that are frequently not represented in test data generated by using the other test techniques. Random testing is performed to detect outputs from the random inputs that deviate from known or expected output values. Test personnel perform random testing of code samples using the appropriate documentation and specifications. Figure D-7 illustrates the process.

Depending upon the knowledge of the input requirements, random testing can be performed with different views of the input distribution. If no data are available concerning the frequency of variables taking on certain values or ranges of values in the operational environment, a test data generator is usually constructed with each variable having an equal probability of being generated for a given test case. However, if the operational input profile is known, the random test data generator can be designed to weight probabilities of variables taking on values or ranges of values, according to the operational input profile. Random testing under an operational input profile attempts to exercise the code as it will be used in operation. A random test case generator can also be designed with a "switch", which either turns on or off the generation of test cases containing only valid versus erroneous data values, to facilitate testing of error handling capabilities of the code under test.

Process

Random (or statistical) testings consists of randomly choosing test cases as subsets of all possible input values according to a uniform probability distribution over the range of values specified for each input. Working from code sample specifications which identify all valid inputs and outputs, the testers code a test generator routine that randomly selects inputs. This generator is then executed to provide test case inputs. Testers determine the corresponding correct outputs expected. Then test cases are prepared from these test input and output pairs, and a test procedure is written to execute them.

The test driver is tailored as needed and a support tool such as DTM is set up to run the test as a test collection. The sample is instrumented with a test tool such as RXVP-80 in order to gather path coverage information. The tested executes the code with the given test case inputs by running the test collection, and uses the support tool to find differences in actual and expected outputs. SPRs are logged for all errors found. If the stopping rule has not been met, the tester returns to the static activity of creating more test cases to achieve the required MTTF of 10 input cases.

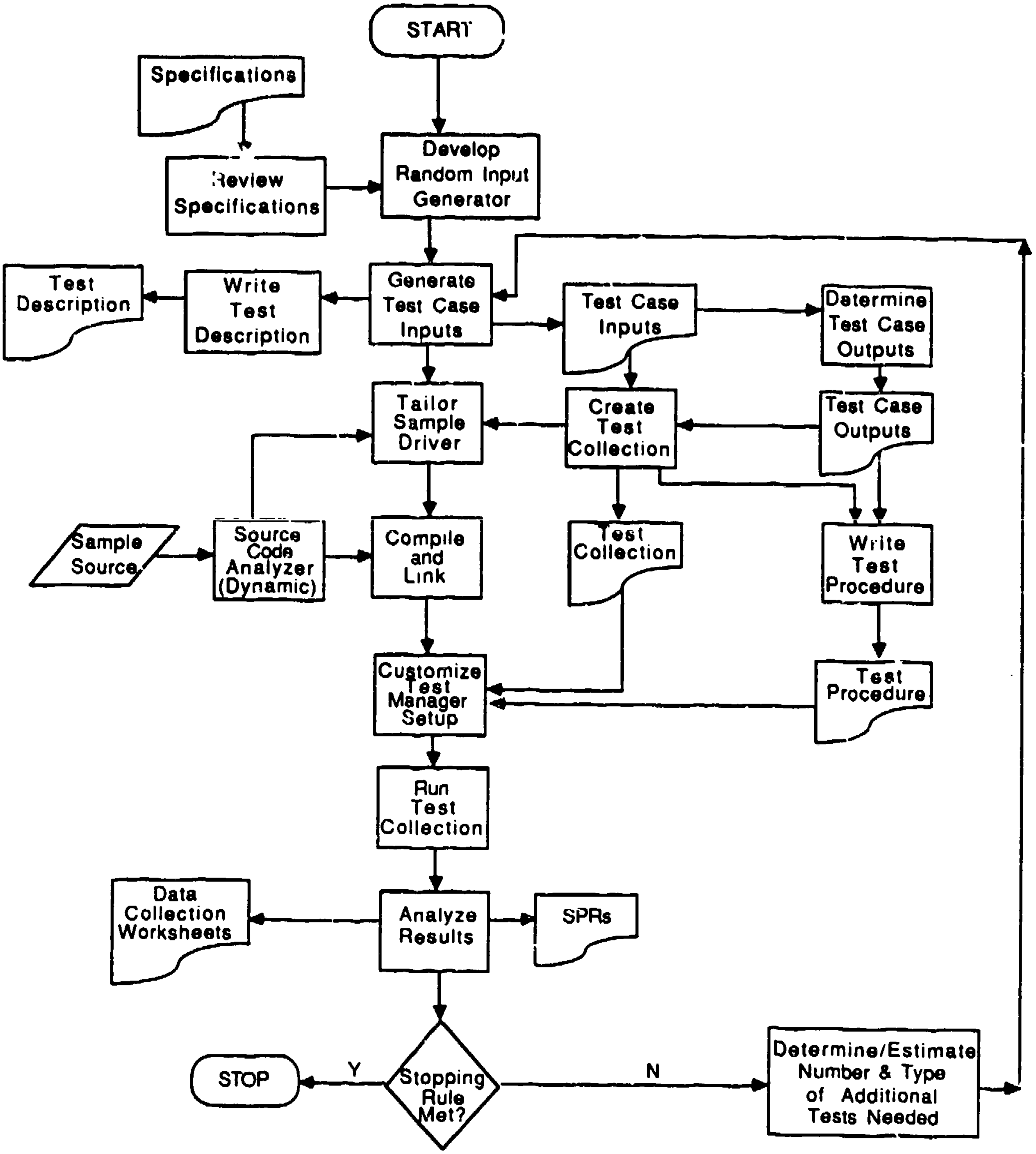


Figure D-7 Random Testing Activity Flow

Information Input

Inputs to functional testing are: 1) the source code of the sample, 2) as abstract specification, or equivalent. of the sample functions, including range specifications of inputs and outputs, and 3) test tool and support tool.

Information Output

Outputs are: 1) a set of test cases and corresponding test procedure(s), 2) a completed SPR for each error observed, and 3) test activity worksheets.

D.4 Test Tools

The best tool available to support and/or automate a test techniques is desired. The test tools identified in Table D-1 are recommended, based on actual use and on inputs from test experts familiar with commercial tools. Any comparable tools to the types discussed here are appropriate. All three are commercially available. RXVP-80 is FORTRAN specific. A comparable source code analysis tool for Ada is the Automated Test and Verification System (ATVS) available from RADC.

Table D-1. Software Test Support Tools

TOOL	APPLICATION	SOURCE
DTM	Manage software samples, test cases, test data.	DEC
SDDL	Support code reading.	SAIC
RXVP-80	Support error and anomaly detection, structural analysis, and branch testing. Also used to provide path coverage information.	GRC

D.4.1 DEC Test Manager

The DEC Test Manager (DTM) is used to organize online test information for the three dynamic testing techniques. Within the DTM testers define test descriptions; each test description defines one test, by associating together the appropriate test sample, its ancillary data files (if any), the sample driver, input cases(s), expected outputs (the benchmark), and actual test outputs. One or more test descriptions are organized into a DTM test collection. The DTM allows test execution at the test collection level. All tests for the dynamic test techniques are run as DTM test collections. The DTM automatically stores test outputs and compares then with expected output (the benchmark), flagging all mismatches between actual and expected outputs.

D.4.2 SDDL

The Software Design and Documentation Language (SDDL) is used in conjunction with the code review technique only. All code samples are run through the SDDL tool prior to the beginning of testing. Each tester uses the printed outputs of SDDLs static processing of the source code. These outputs consist of:

a. Table of contents for SDDLs output.
b. Source code, enhanced with indentation based on structure and with flow line arrows in a two-dimensional representation.
c. Module cross-reference listing.
d. Module reference tree.

D.4.3 RXVP-80

RXVP-80 test tool combines static and dynamic analysis features. RXVP-80 is used to automate Structure Analysis and Error and Anomaly Detection, to process code samples to identify branch coverage for the branch testing technique, and to instrument samples for path coverage information for all three dynamic techniques. Testers use the static features of RXVP-80 to perform static analyses of the code samples as required for error and anomaly detection, structure analysis, and branch testing. First you instruct RXVP-80 to instrument the code under test with RXVP-80 path coverage commands to create an instrumented version of the source for use in dynamic testing. Then you invoke the static analysis capability of RXVP-80 to obtain reports for use with the static analyses of code samples. These operations are performed by running RXVP-80 independently of the DTM, whereas all dynamic testing takes place under the organization of the DTM. Thus code samples which are instrumented with the dynamic component of RXVP-80 can be invoked from within a DTM test description template file.